Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Wenhan Huang

Entitled

PARALLELIZED RAY CASTING VOLUME RENDERING AND 3D SEGMENTATION WITH COMBINATORIAL MAP

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

Dr. Paul Salama
Chair

Dr. Maher Rizkalla

Dr. Lauren Christopher

Dr. Kenneth Dunn

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Dr. Paul Salama

Approved by: Dr. Brian King 4/25/2016

Head of the Departmental Graduate Program Date

PARALLELIZED RAY CASTING VOLUME RENDERING AND 3D SEGMENTATION WITH COMBINATORIAL MAP

A Thesis

Submitted to the Faculty

of

Purdue University

by

Wenhan Huang

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2016

Purdue University

Indianapolis, Indiana

ACKNOWLEDGMENTS

First, I would like to thank Dr. Paul Salama for guiding and helping me to complete this study.

I would like to thank my family, especially to my mother Yuwen Sun, my father Xiang Huang, my wife Qiaoqiao Wan, Kaka, Cindy, Ivan, and Pebbles for their spiritual and financial support to complete this study.

To Dr. Kenneth W. Dunn for his support of this study.

To Dr. Lauren Christopher, Dr. Maher Rizkalla, Dr. Brian King, and Dr. Yaobin Chen for advice and guidance.

To Jerry Su for his contribution on Voxx 3.

To Sherrie Tucker and Jane Simpson for study advising.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

SYMBOLS

V_d	Direction Vector (Point from Camera Location to Voxel Coordinate)
V_{coor}	Voxel Coordinate Vector
V_{camera}	Camera Location Vector
V_{step}	Step Vector (Increment Distant Vector)
V_{scale}	Scale Vector
dt	Period Coefficient
α	Transparency Coefficient of Current Voxel
C_s	Color of Sample Voxel
C_o	Final Pixel Color Observed by the Viewer
S_{block}	Number of Threads per Block
N_{reg}	Number of Register per Block
N_{rpt}	Number of Register per Thread
N_{tmax}	Maximum Simultaneous Threads
N_{data}	Total Number of Data that Needs to be Processed
N_{block}	Total Block Number

ABBREVIATIONS

TIFF	Tagged Image File Format
GPU	Graphics Processing Unit
CPU	Central Processing Unit
RAG	Region Adjacency Graph
OBG	Oriented Boundary Graph
CT	Computer Tomography
PET	Positron Emission Tomography
MRI	Magnetic Resonance Imaging
n-D	n-Dimension
GPGPU	General-Purpose Graphics Processing Unit
SIMD	Single Instruction Multiple Data
CUDA	Compute Unified Device Architecture
SM	Streaming Multiprocessor
API	Application Programming Interface
LD/ST	Load and Store Unit
CC	Computation Capability
SFU	Special Function Unit

ABSTRACT

Huang, Wenhan. M.S.E.C.E., Purdue University, May 2016. Parallelized Ray Casting Volume Rendering and 3D Segmentation with Combinatorial Map. Major Professor: Dr. Paul Salama.

Rapid development of digital technology has enabled the real-time volume rendering of scientific data, in particular large microscopy data sets. In general, volume rendering techniques project 3D discrete datasets onto 2D image planes, with the generated views being transparent and having designated color that is not necessarily "real" color.

Volume rendering techniques initially require designating a processing method that assigns different colors and transparency coefficients to different regions. Then based on the "viewer" and the dataset "location", the method will determine the final imaging effect. Current popular techniques includes ray casting, splatting, shear warp, and texture-based volume rendering. Of particular interest is ray casting as it permits the display of objects interior to a dataset as well as render complex objects such as skeleton and muscle. However, ray casting requires large memory and suffers from longer processing time. One way to address this is to parallelize its implementation on programmable graphic processing hardware. This thesis proposes a GPU based ray casting algorithm that can render a 3D volume in real-time application.

In addition, to implementing volume rendering techniques on programmable graphic processing hardware to decrease execution times, 3D image segmentation techniques can also be utilized to increase execution speeds. In 3D image segmentation, the dataset is partitioned into smaller sized regions based on specific properties. By us-

ing a 3D segmentation method in volume rendering applications, users can extract individual objects from within the 3D dataset for rendering and further analysis. This thesis proposes a 3D segmentation algorithm with combinatorial map that can be parallelized on graphic processing units.

1. INTRODUCTION

1.1 Introduction of Volume Rendering

The invention that has a significant influence on the history of technology development can be classified into two types. The first catalog includes the inventions like the television and the telephone, which changed the livelihood of human firstly and then changed the way of how people see the world. The another catalog does it reversely, like the Relativity and the Roentgen Ray. They first changed the way how people see this world and then changed the way how people live. Volume rendering technology should belong to the second type. The considerable importance of volume rendering is its application in scientific data visualization. One of its primary application is contributed from the rapid development of medical imaging like microscopy imaging, Computer Tomography (CT), and Magnetic Resonance Imaging (MRI), Moreover, volume rendering technology can be applied in geological exploration, weather analysis, molecular modeling and other scientific fields. Right now, volume rendering with 3D image data set has become a major research area in clinic and academy.

The 2D image generated by volume rendering algorithm are transparent, and the color is designated instead of its real color. Volume rendering initially requires classifying processing method, which assigns different regions with different color and transparency coefficient. Then based on the viewer and the date set location, the algorithm will determine the final imaging effect. The current popular algorithm includes ray casting, splatting, shear warp, and texture-based volume rendering. Volume rendering techniques are designed to display the object inside the dataset, so that it can render complex object located in the dataset like skeleton and muscle. However, the drawbacks of the algorithm are large memory requirement and longer

processing time, which makes this algorithm not appropriate for real-time processing. Therefore, the current studies focus on speed up the rendering by parallelizing rendering algorithms on programmable graphic processing hardware or preprocessing dataset to skip objects in the dataset.

1.2 Volume Rendering and Scientific Visualization

Scientific visualization is a technology and methodology of computing. It utilizes the knowledge in computer graphic, image processing, and computer vision to transform the symbolic and digital information into geometry, allowing the researchers to observe the computation result. It can bring enormous advantages to support scientific productivity and the potential for scientific breakthrough [1].

Volume rendering is a sub-branch in scientific visualization. It is a technique used to generate a 2D projection image from a 3D data set. Compare to other scientific visualization technique, The ultimate goal of Volume rendering is to display the 3D detail on an imaging plane. For example, if a house is used as a 3D volume, inside the house, there are furniture, appliance, and other objects. When the users view the house from the outside, only the exterior appearance can be observed, and the layout of the room or the arrangement of objects are occluded; but if the house and the objects are semi-transparent, all the detail can be viewed at the same time. This rendering effect is what volume rendering technique wants to achieve.

1.3 Ray Casting and Parallel Computing

The ray casting rendering technique [2] is the most well-studied image based volume rendering method, due to its high-quality result and capability to render transparent effect. The idea of ray casting algorithm is very similar to light transmission in the real world, but it is done reversely. In nature, a light source emits a cluster of light rays that pass through a transparent object. Some of the rays will fall into out

eye and image on our retina. In volume ray casting algorithm, rays are cast from the eye to an image plane, one per pixel, and find the intersection the objects blocking the path of each ray. A ray is terminated if no other object falls in its path. During the transmission in the object, color values are sampled at a certain interval along the rays.

The limitation of ray casting algorithm is its massive computation. For a 512^3 image, it requires 134,217,728 arithmetic operations to generate each frame. Fortunately, the advantage of this algorithm is the parallel nature in volume rendering. Therefore, ray casting is studied on its acceleration on special rendering hardware. Before GPU became capable of handling this kind of task, rendering hardware like volumePro [3] is widely used in this field. Recent research tends to accelerate the volume rendering technique like ray casting on modern graphic processing unit.

In the 1990s, people started to realize the power of parallel processing and began to perform general purpose computing on the graphic processing unit (GPU) programmable shader. With the development of hardware, GPU shader is capable to read randomly from video texture memory based on mathematical and logical computation [1]. This advancement gives research opportunity to parallel any algorithms such as ray casting and significantly improved the rendering speeding of such method.

1.4 History of 3D Volume Rendering

In the 1970s, The first of volume rendering technique is implemented at Mayo Clinic. Later in the 1980s, due to the advancement of image processing hardware and data manipulation technique, the algorithm of volume rendering can be implemented on the parallel processing system. The volume rendering algorithm is first parallelized at the University of North Carolina and Pixar. The work at Pixar was derived from the work of Ed Catamull, PhD, and Alvey Ray Smith, PhD at LucasFilms, which is

used to generate more realistic computer graphics for the movies. The 3D rendering work in 1990s are significantly contributed by the graph acceleration hardware developed by Silicon Graphics at Mountain View, California. Software that developed at Silicon Graphics optimize its application on medical aspect using special texture mapping technique [4].

In the past, Voxel-based volume rendering software like VolumePro were usually run on very expensive SGI workstations. In the 2000s, graphics processing units can achieve rendering speed at a lower cost. Therefore, many research organizations started to develop volume rendering software products. These software includes the Voreen developed by University of Munster, VolView developed by Kitware, and Voxx developed by Indiana University. These volume imaging programs are designed to use the commercial graphics processors like GeForce and Radeon. However, these volume rendering software do not fully unitize the capability of GPU to perform 3D image processing. Consequently, the new version of Voxx can perform some real-time image processing algorithms like filtering, image registration, and segmentation during rendering.

1.5 Introduction of Image Segmentation

Image Segmentation is a subtopic of the field of Computer Vision, which deals with the partition of a digital image into multiple segmentation. In particular, The goal of this image processing technique is to cluster pixels into salient image regions based on the characteristic. Through simplify the region or change the representation of an image, it transfers the original image into something that is easier to recognize by the computer or human eyes for further analysis. The segmentation algorithm is usually a middle step of an image processing algorithm such as object detection, recognition, image compression, and occlusion boundary estimation. Therefore, segmentation technique has a wide application in the vast field of Artificial Intelligence.

1.6 Segmentation Method

Based on application, there are many different approaches to performing image segmentation. Some popular methods are listed below.

Threholding

Thresholding method is one of the most popular segmentation algorithms due to its idea simplicity and fast speed. A typical thresholding segmentation algorithm uses a threshold value to convert a gray-scale image into a binary image. This method relies on selecting the appropriate threshold value. Since its advantage in processing speed, this approach is implemented in many image processing software.

Edge Detection

Edge detection algorithm uses the sharp pixel intensity adjustment at the region boundaries to classify different objects. In edge-based segmentation algorithm, a sharpen filter is applied to the image, which is finding the derivative of a 2D signal in mathematics. The pixels that are not separated by an edge are classified into the same region. The output edge image generated by the sharpen filter are usually rough edges, which means the edge is too wide to distinguish the region. Some edge thinning methods are applied to the edge image.

Region-Merging

Region-Merging segmentation compare every pixel inside a image with its neighbors using some merging criterion. Methods rely mainly on the assumption that the neighboring pixels within one region have similar features. The common procedure is to compare each pixel with its neighbors. If a similarity criterion is satisfied, the pixel will be set to the cluster as one or more of its neighbors. The selection of the similarity criterion is significant due to the results are influenced by noise.

Watershed

Watershed segmentation uses the gradient magnitude in an image as the separate line between different region, which is called as the region boundary lines and watershed lines. To find the line in an edge image, the algorithm traces the pixel values with the highest gradient value. Then assume water is added into the image and pixels that belong to the same catch basin form a segment.

1.7 Organization of the Thesis

This thesis is organized as follows. The first part (chapter 3) describes Voxx 3, a real-time volume rendering software for PC. Voxx implements the ray-casting algorithm with parallel voxel by voxel sampling. The software can render more than 1000 512×512 TIFF images with 500 samples per ray at 30 frames per second using graphics processing units (GPU) pipeline. In this thesis, the software architecture is discussed, which focusing on the software decomposition, OpenGL rendering algorithm, sampling method, and user input parameter. Then several features of Voxx 3, such as a palette module that allows user modify rendering result, the image and video capture module, and the image processing algorithms module are presented.

The second part (chapter 4) presents an algorithm to extract a topological model for a given 3D image that represents both geometrical and topological information. The calculation of the minimal topological map of a 3D image occurs in three stages. The output of each stage is the input of next stage. The outcome of each stage represents the intervoxel boundary information of surface elements, edge elements, and vertex elements correspondingly. The combinatorial map is the last stage in the hierarchy. In the second section, the definition of the combinatorial map and its applications and encoding method is given. Then a sequential algorithm that extracts the map of a given image in $O(n_3)$ time and a discussion about how to use this map in the 3D segmentation algorithm is provided.

The last part (chapter 5) discusses the background knowledge of GPU algorithms based on CUDA. GPUs are widely used in image processing application. This section reviews NVidia Fermi GPU architecture, CUDA programming model, and CUDA Kernel optimization scheme. Finally, the combinatorial map extraction algorithm is modified and optimized for the general CUDA kernel.

2. VOXX 3

2.1 Purpose and Background

Volume rendering has wide application in scientific data visualization, especially in medical imaging. Medical images obtained via X-ray Computer Tomography (CT), Positron Emission Tomography (PET), Magnetic Resonance Imaging (MRI) and Microscopy are usually composed of stacks of images T different depths or time points. Therefore, constructing of volume view instead of multiple single 2D views is crucial for researchers to analyze data, extract information, as well as helping physicians in performing diagnosis. Thus, the utility of volume rendering software is significant.

Ray casting [2] is very popular in volume rendering due to its conceptual simplicity and high-quality result. However ray casting rendering algorithms require much computation for each ray resulting in longer times to generate every frame. The computation requirements make ray casting not practical in real-time rendering using serial computation. In order to improve rendering speed, one approach optimizing techniques that involve preprocessing the dataset are used to address the problem [5–7]. One approach involves skipping voxels or regions can be skipped during rendering. This technique requires repeated preprocessing after changing rendering parameters such as the voxel transparency coefficient. This technique also requires extra memory to store preprocessed dataset. Alternatively, [8, 9] described preprocessing methods that utilize segmented object rendering and iso-surface rendering, while [10] proposed an object order algorithm that terminates ray earlier in order to speed up rendering.

Moreover, special hardware that is designed for accelerating ray casting algorithms has been used to render volume [3, 11–13]. Sampling and composting in the dataset

are performed on optimized hardware increase the rendering speed. [14] proposed using texture mapping hardware like Silicon Graphics RealityEngine to render volume data. The dataset is stored in 3D texture memory, and samples are extracted from texture planes parallel to the image plane. This approach requires customers purchase expensive hardware. Instead of using specially designed hardware, a graphical processing unit (GPU) accelerated real-time rendering software, Voxx 3, is proposed in this thesis. Voxx 3 samples volume data that are stored in GPU texture memory using direction gradients that calculated from texture coordinates and camera coordinates.

2.2 GPU Graphics Pipeline

2.2.1 OpenGL

OpenGL is a cross-platform and cross-language application programming interface that operates graphic processing unit to render a real-time 2D image on the computer screen. It has wide applications in game development, CAD, and data visualization. OpenGL started as a cross-platform standardization developed by Silicon Graphic Inc for their workstations in the 1990s. Until the late 1990s, OpenGL is accepted by the massive developer and then it become a standard graphic library for operating OpenGL graphic processing unit in PC [15].

2.2.2 Rendering Pipeline

GPU rendering pipeline performs a series of processing stages in order. In the pipeline, The output of one stage is the input of the next stage. In Figure 2.1, vertex data are processed through the pipeline and the result are written into the screen buffer.

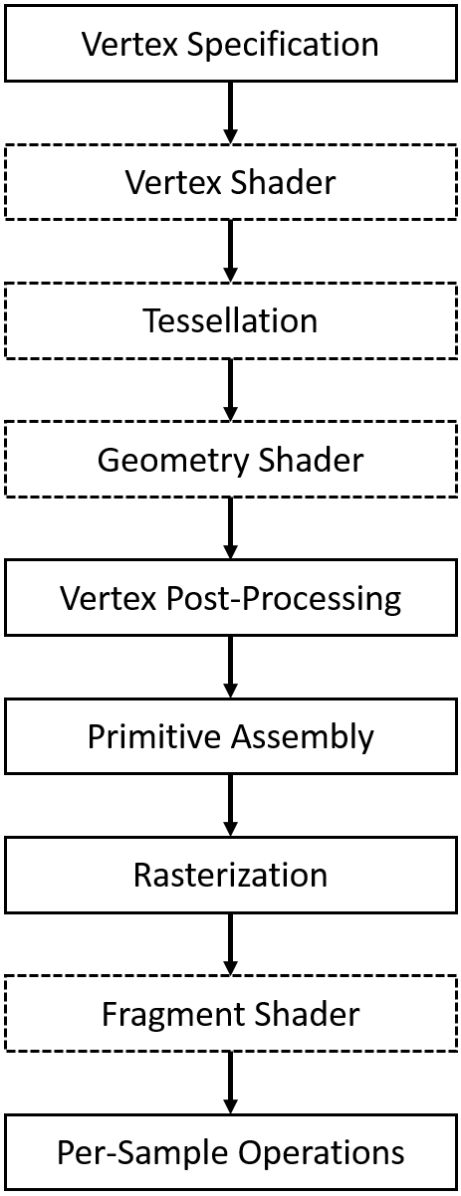


Fig. 2.1.: Diagram of the Rendering Pipeline. Dot Block are Programmable Stage.

Vertex Specification

In the vertex specification stage, a list of vertices with information like vertex location in 3D space, corresponding texture coordinate, plotting index, and vertex color are send into the pipeline. Generally speaking, the graphic pipeline starts with the 3D model; this model can be a 3D game character designed used modeling

software, or the vertices generated by a laser scanner. These vertices, which define the approximate boundary of objects, are stored in the GPU buffer such as Vertex Array Objects and Vertex Buffer Objects. In this step, transformation operation like rotation, scaling, translation can be applied to the vertices.

Vertex Shader

The process of GPU rendering pipeline begins with loading vertex attributes that stored in Vertex Array Objects or Vertex Buffer Objects and passing through the vertex shader. If a user defined vertex shader is not provided, GPU pipeline will pass vertices into a default shader. In this shader, GPU calculates the projected position of each vertex in array buffer and mapping them into screen space. Color or texture coordinates also can be modified based on the users' demand.

Tessellation Shader

Tessellation shader is an optional stage in the GPU rendering pipeline. Until this stage, vertices can be operated using the vertex shader. Even though vertex shader is powerful, it still has a limitation, which is incapable of creating new primitives. To solve this problem, tessellation shader adds a new primitives types to the existed vertex array, called the patch. A patch is a primitive with vertices that defined by the user. The tessellation shader divides a patch into vertices and forms a triangle mesh.

The processing of tessellation consists of three stages, which are tessellation control shader, tessellation primitive generator, and tessellation evaluation shader. The first stage and last stage are programmable; the middle one is a fixed function stage [16]. The tessellation control shader determines the amount of tessellation for each primitive and performs designated transformation on the patch data.

Geometry Shader

The output vertices generated by the tessellation shader are then passed into an optional shader stage, called geometry shader. The geometry shader uses the single primitive as the input and transforms them into completely different primitives.

Vertex Post-Processing

After shader based vertex processing, the vertices are passed into a fixed post-processing stage. Depends on the projection method and view size, the projection matrix is calculated and primitives are clipped.

Primitive Assembly

In primitive assembly stage, GPU connects the vertices generated from previous stages. The GPU takes the vertices in the order specified by vertex array or user and segments them into triangles. Current rendering pipeline supports three methods [15].

1) Take every three vertices inside vertex array buffer. This method requires $3 \times n$ vertices to generate n triangles.

2) Triangle strip: A triangle strip is a list of triangle vertices, which each triangle shares an edge with the previous triangle. an example is shown in Figure 2.2, A triangle strip or triangle fan are commonly used to compress the number of vertices that form an object.

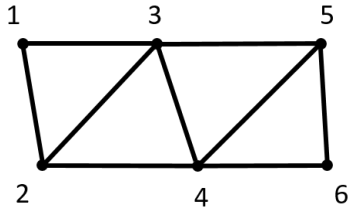


Fig. 2.2.: A Triangle Strip.

3) Triangle fan: Triangle fans uses the same idea of triangle stripe. In a triangle fan, a vertex is shared with other two vertices where each three vertices will form a triangle. Figure 2.3 shows an example of triangle fans.

The indexes in Figure 2.2 and Figure 2.3 indicate the order when plotting each

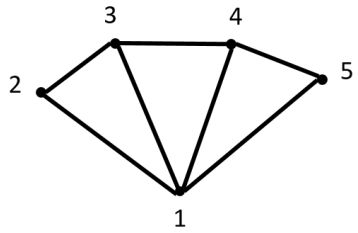


Fig. 2.3.: A Triangle Fan.

vertex. In Figure 2.2, edge 2-3 is shared with the first triangle and second triangle. When GPU plots the triangles in the triangle strip, it takes the previous two vertices in the vertex array with a new vertex to generate each triangle. Compare to the first method, triangle strip requires $n + 2$ vertex to represent n triangles. In Figure 2.3, Vertex 1 is shared with all the triangle. When plotting, GPU takes the first vertex and fetch two vertices in the vertex array to generate triangle.

Rasterization

Rasterization refers to the process of clip each primitive that fall outside the screen, break down the residual primitive into pixel size fragments, and assign pixel color to the fragments based on the vertex texture coordinate. In this step, GPU determines the location of the display window coordinate that occupied by the fragments and assign depth value as well as texture coordinate to each fragments. The generated fragments are sent into the next stage.

Fragment Shader

A fragment shader is the last programmable shader in rendering pipeline. The fragment shader operates fragment color based on user input, texture mapping, and lighting. The operation of on each pixel in the fragment runs independently from others. Therefore, fragment shader can provide the most impressive effect.

Per-Sample Operation

Per-sample operation stage contains a serial test for the fragment data generated by the fragment shader. The tests include Pixel ownership test, scissor test, stencil test, and depth test. These tests can be either turn on or disable. At last, the fragment data is written to the frame buffer.

2.3 Architectural Design

The Voxx 3 consists of several modules: a 3D stack image module, an OpenGL rendering module, an OpenGL parameter module, a color look-up table module, an image capture module and a movie record module. The interconnection among these modules provides the complete functionality of Voxx 3. Figure 2.4 describes the relationship between modules.

The 3D stack image module and the OpenGL rendering module are the core modules of the program. They achieve the basic functionality of the program. The 3D stack image module decodes the input images and stores data in a 2D array in memory, which is easier to use by the OpenGL rendering module. The OpenGL rendering module controls the GPU graphic pipeline, and responsible for displaying the rendering result on the screen. To change the rendering result based on user input like rotate or increase region intensity, user input parameter must be sent into OpenGL rendering module. Therefore, a OpenGL parameter module and color look-up table

module is designed to take user input. These two modules convert the user input to the parameter that can be used in OpenGL. Specifically, the OpenGL parameter module allows user to change blending method, modify volume boundary, and convert mouse movement into rotation angle. The look-up table module allows users to modify the color of rendering result, through modify the corresponding look-up table, where image voxels are used as indexes and final voxel values are find in look-up table. More detailed information can be found in section 2.4.5 and section 2.4.6.

In order to support function like region selection and improve rendering result, a image processing module is designed to support various image processing algorithm. The image processing module modifies voxel value that stored in 3D stack image module. Other features are implemented using the remaining modules. For instance, Image capture module and movie record module transfers pixel value in screen buffer back to memory and encodes it to images or movies for better demonstration purpose.

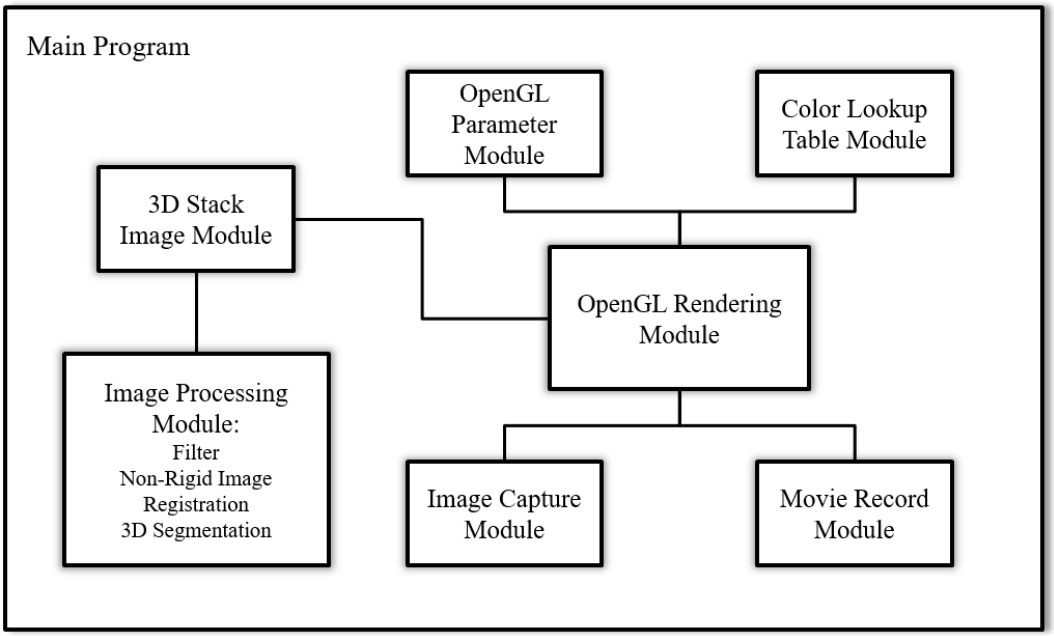


Fig. 2.4.: Voxx 3 Software Module Decomposition.

2.4 Decomposition Description

2.4.1 OpenGL Rendering Module

The rendering module utilizes volume ray casting, an image-based rendering technique, to create a 3D perspective from a 3D texture data set.

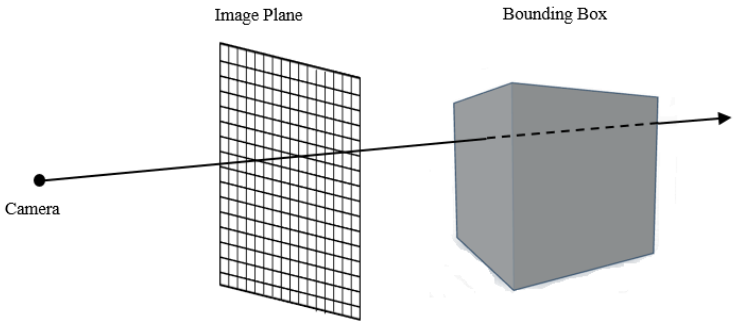


Fig. 2.5.: Ray Casting.

Figure 2.5 shows a diagram that illustrate the whole process of ray casting algorithm. To set up the sense in openGL world coordinate, a camera, a image plane and a bounding box are created. Ray casting consists of three steps to generate result on image plane [2]:

1) Ray casting:

For every pixel on the screen, a ray is sent out, starting at the camera location, with a direction vector depending on the position of volumetric data location.

2) Sampling:

Along a ray cast through the volume, it is possible to sample the voxels for a certain interval. For the sample points that located between voxels, linear interpolation is applied to estimate voxel values from neighboring voxels.

3) Shading and compositing:

After interpolation, a pixel (Pixel is used when dealing with 2D images and Voxel is used for 3D iamge) value is calculated for each sample point on the ray. Then based on the shading method, all the values of sample points are composited together using

a corresponding blending equation, resulting in a final pixel color. Eventually, this final pixel color is stored in the screen buffer and displayed on the screen.

Sampling in Normalized Dimensions

To represent a ray, direction vectors are obtained by calculating the casting direction of ray from the coordinate of the camera to the coordinate of image plane. Equation 2.1 shows how to calculate normalized ray casting direction V_{dir}. In the thesis, (x, y, z) represents the coordinate of each vector. In the equation, V_{camera} represents the camera location in OpenGL world coordinate and V_{coor} represents the image plane coordinate. Normalize indicates converting a vector into a unit vector.

$$V_{dir} = normalize(V_{coor} - V_{camera}) = \begin{pmatrix} x/\sqrt{x^2 + y^2 + l^2} \\ y/\sqrt{x^2 + y^2 + l^2} \\ l/\sqrt{x^2 + y^2 + l^2} \end{pmatrix} \tag{2.1}$$

After direction vector is obtained, the next step is acquired intersection point of casting ray and bounding box [17]. A ray can be represented using the camera coordinate and ray direction. Equation 2.2 shows the mathematical representation of a casting ray. In the equation, dt is a distance value that is used to calculate any point on the ray.

$$V_{ray} = V_{camera} + dt \times V_{dir} \tag{2.2}$$

All of the samples points that are used to calculate corresponding output pixel color fall along on this ray. The next step is to calculate step vector. This is necessary if the data set does not have same dimensions. Because to utilize date set in OpenGL, 3D data set must be transferred into texture memory. After images are transferred into texture memory, voxels are accessed using texture coordinates instead of its original indices. OpenGL texture coordinate is normalized, which means for a 3D

texture, its front left bottom corner is mapped to (0,0,0) and back top right corner is mapped to (1,1,1). Color information is retrieved from the voxels using these texture coordinates. As illustrated in Figure 2.6, it shows how to look into an object in the normalized coordinate. The small box represents the actual size of a 3D data set and the large box represents the box that is extended into the same dimension. Therefore, a step vector V_{step} must be calculated to sample in the normalized coordinate.

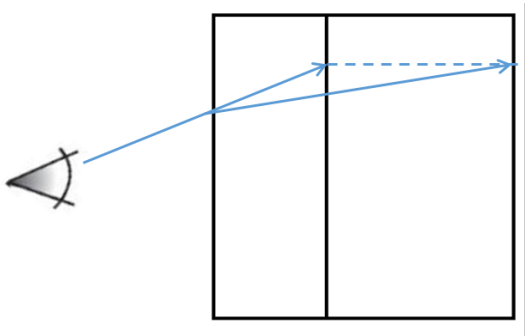


Fig. 2.6.: Ray Casting into Normalized Coordinate.

Step vector should contain information about the direction and distance of the direction vector. Since different data sets may have different dimension, every component in step vector should increment different distance during sampling. To calculate step vector from direction vector, a weight value C_{weight} is given to every component in the step vector based on its corresponding dimension. C_{weight} contains the weight coefficients for each coordinate. The equation 2.3 show how to calculate step vector. In the equation, d_x, d_y, d_z indicates the corresponding dimension of a 3D stack image. $max(d_x, d_y, d_z)$ means to choose the maximum value between d_x, d_y, d_z.

$$C_{weight} = \begin{pmatrix} max(d_x, d_y, d_z)/(d_x) \\ max(d_x, d_y, d_z)/(d_y) \\ max(d_x, d_y, d_z)/(d_z) \end{pmatrix} \tag{2.3}$$

Then this weight coefficients is multiply with V_{dir} to calculate the step vector as shown in equation 2.4.

$$V_{step} = C_{weight} \times V_{dir} \tag{2.4}$$

Equation 2.5 gives the method to sample in the texture memory. $V_{current}$ represent the current texture location. This value is calculated based on the intersection between ray and bounding box [17]. $V_{coor\,next}$ the the next texture coordinate that need to be sampled. For any given interval dt, sampled voxels in the texture are saved and blended using the method purposed in next section.

$$V_{coor\,next} = V_{current} + dt \times V_{step} \tag{2.5}$$

Ray and Bounding Box Intersection

The middle step of a ray casting algorithm is to test the intersection between bounding box and each ray generated by the camera and image plane. The algorithm of performing ray and bounding box intersection is the one proposed in [18]. This algorithm tests the intersection between a ray and an axis aligned bounding box. An axis aligned bounding box indicates the faces of a bounding box must parallel to the coordinate plane. The algorithm treats the faces of the box as the planes that parallel to coordinate plane. The ray is clipped by the pair of parallel plane, and the portion inside the box intersected the box.

Figure 2.7 shows an example of testing intersection points between a ray and an axis aligned bounding box. It is easy to find the intersection point t_{min1} of ray and plane X_1 and intersection point t_{max1} of ray and plane X_2. Then, points t_{min1} and t_{max1} are further tested with plane Y_1 and Y_2. Since t_{min1} is located outside of plane Y_1, this point is replaced with t_{min2}. In the end, t_{min2} and t_{max1} are the intersection points of the ray and bounding box. dt is calculated using $(t_{min2} - V_{camera})/v_{dir}$.

Blending Methods

All the voxels along the ray need to be composed together to obtain a final output pixel color. The method of composing them together is called the blending function.

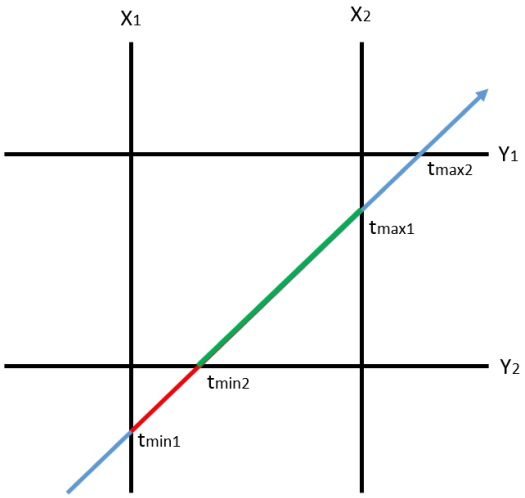


Fig. 2.7.: Testing Axis Aligned Bounding Box and Ray Intersection.

Voxx 3 currently supports three blending methods: maximum, mean, and alpha blending. These three different blending methods will generate different rendering results.

Maximum blending uses the maximum voxel value along the casting ray as the final pixel value, as given by equation 2.5. In equation 2.5, C_s and C_o represents current sample voxel and the output sample voxel correspondingly. The naming method applies to equation 2.6 and 2.7.

$$C_o = \sum_{s=0}^{n} Max(C_s, C_o) \tag{2.6}$$

Mean blending uses the mean value of all the sample points on the casting ray, mean blending method average all the sample points on the ray by adding up all the pixel value and divided by the number of sample point:

$$C_o = \frac{1}{n} \sum_{s=0}^{n} C_s \tag{2.7}$$

Alpha blending uses a transparency coefficient to compose multiple voxels together. An object transparency represents the capability that light transmit through the object without being scattered. When light passes through an object, its wave length will be changed based on the property of the object. When it passes through multiple objects, this change will be accumulated. Therefore, the fundamental of alpha blending is to mix the color of the current voxel with the next voxel on the casting ray. It preserves some features from the current voxel and also includes with part of the features from the next voxel. The equation of alpha blending is given in equation 1.7. In the equation, the α value represents the transparency coefficient of a voxel. C_{s+1} represents the next voxel value and C_o represents the output pixel value.

$$C_o = \sum_{s=0}^{n-1} \alpha_s C_s (1 - \alpha_{s+1}) + \alpha_{s+1} C_{s+1} \tag{2.8}$$

2.4.2 Color Look-up Table Module

A color look-up table is a mapping mechanism that converts a range of pixel value into another range of colors. Look-up table is used to manipulate the image data. So the user can modify the mapping mechanism by modifying the corresponding look-up color to perform different look-up result. In general, look-up table is applied to a gray-scale image. The table converts gray-scale images into color images. In Voxx 3, the color look-up table is used to increase the contrast of the region of the rendering image. The look-up table in Voxx 3 consists of three channel arrays of integers as shown in Figure 2.8. Original pixel color is treated as an address index to perform look-up in table. Figure 2.7 indicates how to find a corresponding color for a given value f. The red, green, blue, and pink curves in Figure 2.7 represent the output pixel values (For a 8 bits image, the range is from 0 to 255) in red, green, blue, and alpha channels.

For every sample on the casting ray, corresponding output from the sample value is found using look-up table and compose them using blending methods mentioned in

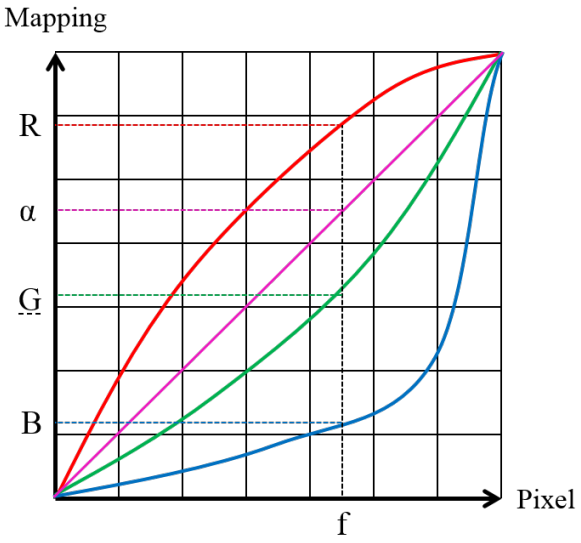


Fig. 2.8.: Visualization of Color Look-up Table.

the previous section. In most color formats, the three channel arrays are either three primary colors, red, green and blue or HSI such as intensity, hue, and saturation. The procedure of color look-up and color composition is illustrated in Figure 2.8. As an example in Figure 2.9, one ray is sampled and the method is applied to other rays. On this ray, sample points 2,3,1,4,5 in red channel, 4,8,9,8,5 in green channel, and 19,7,8,5,8 in blue channel are obtained from dataset. These values are used as the index of color look-up table and the maximum blending method proposed in section 2.4.1, outcome (19, 18, 30) is the final pixel color that stored in pixel buffer.

2.4.3 3D Stack Image Module

The 3D stack image module in Voxx 3 provides the functionality of decoding input images and storing the decoded values. The module is implemented using an abstract class called threeDFiles. The threeDFiles class contains basic 3D image information like stack image dimension, image data pointer, and public functions that allow other modules to write and read these parameters. This class provides data set for the

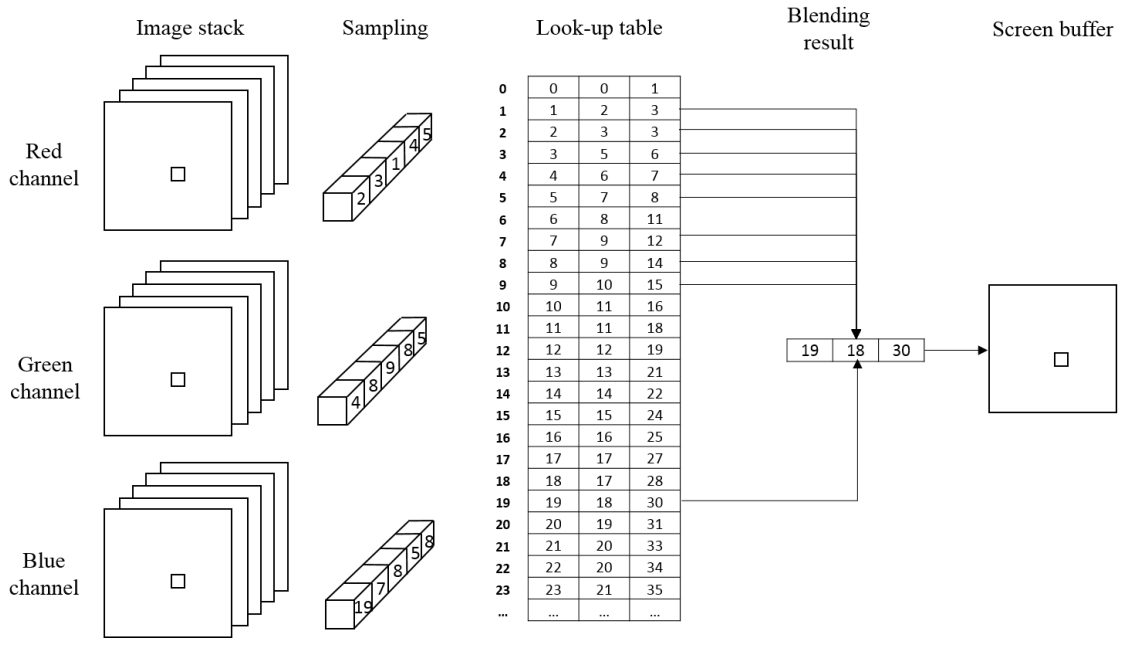


Fig. 2.9.: The Process of Ray Casting Rendering and Color Look Up.

OpenGL rendering module, as well as image processing function module. In order to support different image formats, the threeDFiles class contains a virtual function to decode 3D stack images. This function is declared in threeDFiles class but without implementation. Therefore, this class cannot be instantiated, and it requires its subclasses to implement decoding methods. Voxx 3 currently provides tiff3DFile inherited from threeDFiles to decode input TIFF images.

2.4.4 OpenGL Parameter Module

The functionality of OpenGL parameter module is passing user input and parameters to OpenGL rendering module. It achieves interaction between users and Voxx 3. Voxx 3 allows user to rotate, zoom in, zoom out, scale, cut the object, and select different blending methods. The Figure 2.9 shows the data flow between different modules in Voxx 3. Once the parameter is changed, the updated values are sent to OpenGL rendering module to get new rendering result. OpenGL rendering module

uses this parameter to sample texture data in the 3D stack image module. In the parameter module, a transformation like rotate and scale are performed using matrix operations. In ray casting volume rendering, the rays generated from the camera and image plane are rotated instead of the bounding box. This is because testing intersection between ray and axis aligned bounding box is much faster than testing intersection between ray and bounding box in the arbitrary coordinate. Results of transformation are obtained using an Affine Transformation implemented in homogeneous coordinate system [19].

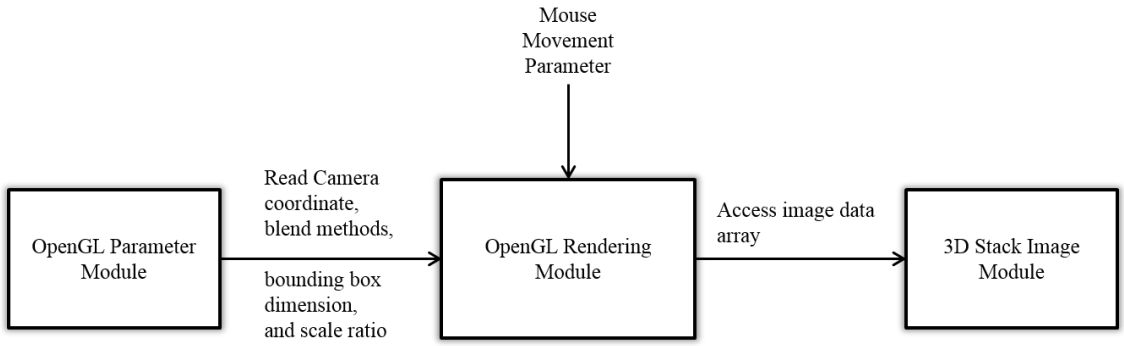


Fig. 2.10.: User Input Parameter and Data Flow Chart in Voxx 3

An Affine Transformation is a linear mapping that transforms an original 3D coordinates to a new 3D coordinates [19]. It preserves the linearity (all points on a line lie on the same line after transformation) and the vertex sequence (the order of each point on a line remains the same after transformation) of the original 3D images. In Voxx 3, a transformation is decomposed into rotation around x-axis and rotation around y-axis. Rotation operation, in 3D case, is to transform the voxel coordinates (x, y, z) by rotating the coordinates round a reference point with an specific angle. This can be achieved through matrix multiplication. Rotate transformation in 3D case can be expressed in matrix form as:

$$P' = Rx(\Theta)P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\Theta) & -sin(\Theta) & 0 \\ 0 & sin(\Theta) & cos(\Theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ \alpha \end{pmatrix} \tag{2.9}$$

Rotation about the y-axis in 3D case can be expressed in matrix form as:

$$P' = Ry(\Theta)P = \begin{pmatrix} cos(\Theta) & 0 & sin(\Theta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\Theta) & 0 & cos(\Theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ \alpha \end{pmatrix} \tag{2.10}$$

In the equation 2.9 and 2.10, P represents either an original vertex coordinate or a vector. P' represent its final coordinate after transformation. Θ is defined as the rotation angle. In Voxx 3, if user rotates the object for 30 degrees, a rotation matrix $Ry(\Theta)$ is multiplied with ray direction vector V_{dir} to obtain the final ray direction vector. As consequence, the intersection between ray and object is changed and so does the rendering result.

$$P' = SP = \begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ \alpha \end{pmatrix} \tag{2.11}$$

Scaling enlarges or diminishes an object by a scale factor S. In equation 2.11, matrix S contains the scale factors Sx, Sy, Sz corresponding to X-axis, Y-axis, and Z-axis. In Voxx 3, if user enlarge or diminish an object, a scaling matrix is mul-

tiplied with the image plane. If scale factor S is larger than 1, then the object is diminished. This is because enlarging the image plane will decrease the density of rays correspondingly. More rays will miss the object and only the rays that pass the image plane center will hit the object. As the result, the object will looks smaller. In contract, if scale factor S is smaller than 1, then the object is enlarged.

Final Transformation is linear combination of rotation matrix and scale matrix can be expressed as:

$$P' = SRy(\Theta)Rx(\Theta)P \tag{2.12}$$

2.4.5 Image Capture Module and Movie Record Module

In Voxx 3, image capture module allows user to save screen display as images. Additionally, for better demonstration effect, the movie record module allows users to rotate the 3D object and record this 3D simulation. To obtain the real-time image or video, Voxx 3 is designed to transfer the pixel values in GPU screen buffer to CPU memory using direct memory access. To improve the capability and easier for user to access, the pixel values can be encoded into different format such as TIFF, PIC, PNG for images and MP4, MKV for videos.

2.4.6 Image Processing Function Module

Image processing function module contains three image processing algorithms including 2D/3D filter, Non-rigid image registration, and region-based 3D image segmentation. This module can visit data pointer in 3D image stack module, modifies voxels using various image processing algorithm and saves the data pointer back to 3D image stack module to improve rendering results or achieve functionality like region selection. This module also contains image processing algorithms that running on GPU. These GPU functions are implemented using CUDA.

1) Filter:

Filter operation is widely used in image processing study. It can suppress or enhance a certain range of frequencies in the image to improve the quality of the image. Image filter can be applied either in frequency domain and spatial domain. To apply image filter in frequency domain, the transformation in frequency domain must be found and multiplied by the frequency function (usually a window function), then transform the result back into spatial domain. Processing image filter in the spatial domain is more widely used, first the inverse Fourier transform of filter function is found, and then convolve with the input image. The output image can be calculated as:
For 2D images:

$$g(x,y) = \sum_{m=-M/2}^{M/2} \sum_{n=-M/2}^{M/2} h(m,n) f(x-m, y-n) \tag{2.13}$$

For 3D image:

$$g(x,y,z) = \sum_{m=-M/2}^{M/2} \sum_{n=-M/2}^{M/2} \sum_{l=-M/2}^{M/2} h(m,n,l) f(x-m, y-n, z-l) \tag{2.14}$$

2) Region Selecting function:

Region selection tools are designed to select regions on the 3D volume that rendered by Voxx 3 so that users can perform any edits, functions, or operations on them without affecting the unselected regions. Operations include analysis information or run image processing algorithm on a user-designated region. In order to select a regions inside of a 3D volume, a 3D segmentation algorithm are designed to partitions volume into multiple regions based on some criteria. Hence, user can select any region based on their demand.

In image processing, segmentation of an image is the process that partitions an image into multiple regions. The purpose of segmentation is to classify pixels with homogeneous properties (pixel values) into the same region for easier analysis and observation. These regions can be tread as different objects that allow users to extract more information. There are many different segmentation algorithms, such as thresholding, clustering, edge detection and region based segmentation. Edge detection [20] method is a well-studied topic in image processing area. Many papers proposed very good segmentation results using this algorithm. It is very popular in image processing area. These algorithms are based on filter operation which is suitable for parallel processing. Because parallel processing is much faster than sequential processing when dealing with large amounts of data, this advantage is significant in 3D image processing. However, the main drawback of edge detection algorithms is that edges are too rough to mark regions boundaries before edge thinning. And edge thinning technique may result in disconnected boundaries. Due to the intolerance of disconnect boundary when region selection (region selection usually uses a start pixel and perform breadth first search (BFS). BFS uses boundary as stop criteria. Therefore, disconnect boundary results in over selection). Therefore, In Voxx 3, region based image segmentation is more stable than other algorithms. Region-based image segmentation requires a representation of region boundaries in the image. As in next chapter, a combinatorial map is very advantageous in representing adjacency relation among regions in a N-dimension image. This representation allows traversing between adjacent regions using different operations, such as find the left region of the current region or find the above region of the current region. In next section. This thesis will further discuss the basic definition of combinatorial map, the operation to construct a combinatorial map, and application in 3D segmentation.

2.5 Contribution

Previous Voxx development has focus on image registration and the rendering module. The present work is designed to provide many of the capabilities of the commercial volume rendering programs, including:

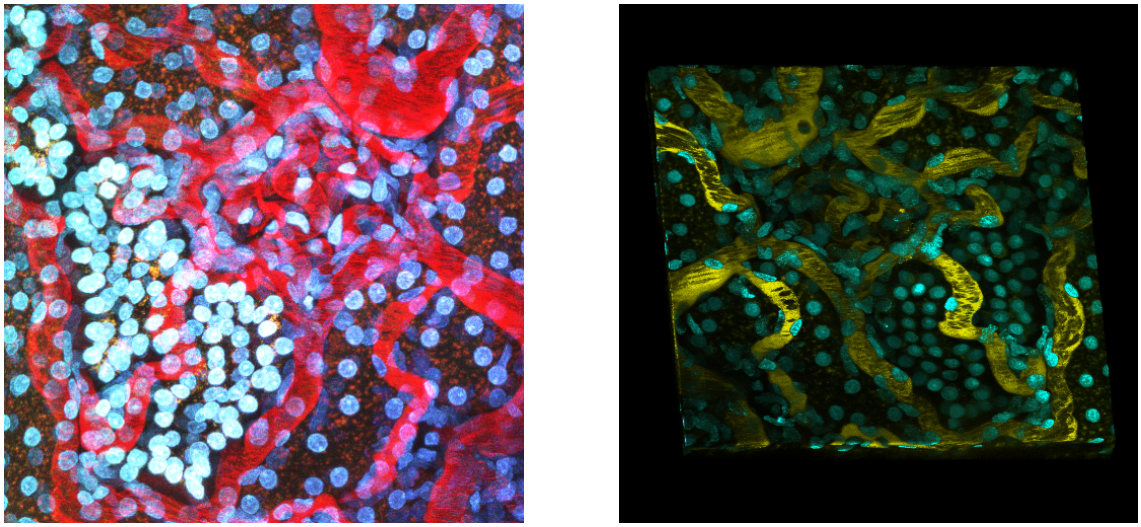
- support 12-bits and 16-bits TIFF images
- support resampling methods and channel skip
- modify graph-based color and opacity table editor
- makes 3D movies(saved as AVI, MP4, TIFF series)
- makes screen snapshot(saved as TIFF, PNG, JPG)
- 2D and 3D filtering and 3D segmentation

2.6 Result

The software is tested on a PC with 64-bit Windows 8 operation system, an Intel Core(TM) i7-4790 CPU @ 3.60GHz, NVIDIA GeForce GTX 760 Ti, and 16GB RAM. An example in Figure 2.1 is the rendering result of a stack image with dimension (512×512×43). The picture is imaged using image capture module. It was rendered with 512×512 pixels, contains 262144 rays at 100 samples per ray.

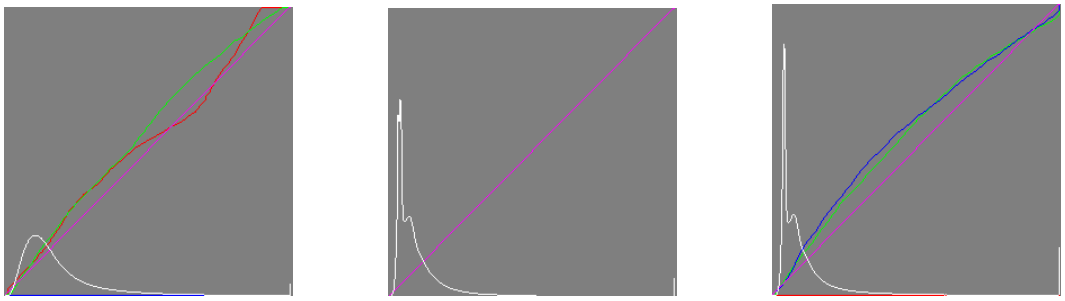
Figure 2.10(b) shows the same image with rotation at angle 350° around x-axis and 200° around y-axis, maximum blending method, sample rate 200, and modified color look-up table.

The three palettes in Figure 2.11 correspond to red, green, and blue channel. In Figure 2.10(b). The X-axis of the palette represents the look-up address index from 0 to 255 for an 8-bits image. Y-axis represents output value from 0 to 255. The red,



(a) Result 1. (b) Result 2.

Fig. 2.11.: Voxx 3 Rendering Result of Microscopy Image Dataset.



(a) Red Channel. (b) Green Channel. (c) Blue Channel.

Fig. 2.12.: Palette of RGB Channel, Look-up Table Visualization.

green, blue (RGB) line in the image represent the corresponding output value from 0 to 255(8-bits image) in RGB channel. The pink line represents the alpha coefficient. The white line is the statistics histogram of pixel values in each channel. As shown in the red palette, red line and green line is modified based on user's demand, blue line remain zero. As the rendering result, the color of blood vessel is changed to gold. Green color is added in blue channel, which resulting the cell color in the image changed to Aqua.

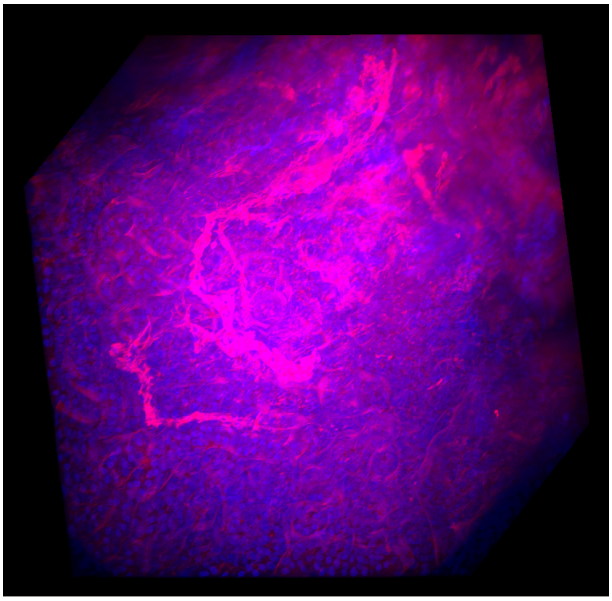
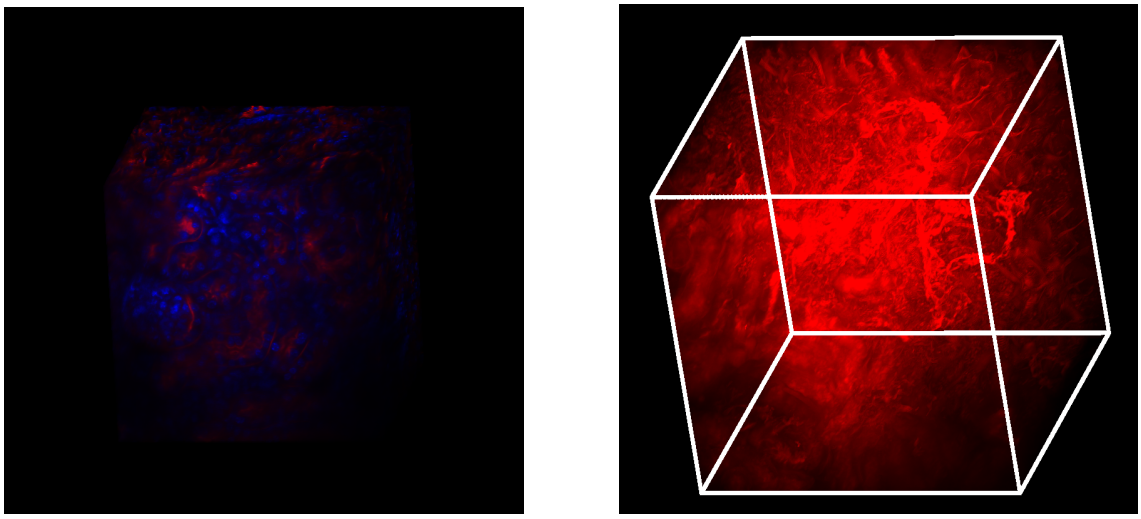


Fig. 2.13.: Microscopy Image Dataset.

Figure 2.12 is a different stack image rendering result tested on a PC with OS 64-bit Windows server 2008, an Intel E3-1225 @ 3.20GHz, NVIDIA Tesla c2075, and 32GB RAM and dataset. Rendering result is sampled from a stack image with dimension (512×512×512). This volume is sampled at 500 samples per ray with maximum blending method.

Figure 2.13(a) is the same volume with maximum blending method but clipped into smaller dimension. Figure 2.13(b) is the same volume with blue channel removed and a bounding box.



(a) Alpha Rendering Result.

(b) Maximum Rendering Result.

Fig. 2.14.: Microscopy Image Dataset with Different Parameter.

3. COMBINATORIAL MAPS AND 3D IMAGE SEGMENTATION

A combinatorial map is a mathematical model that describes the intervoxel boundary and geometrical information. It encodes all the voxels, intervoxels and adjacent relation between them. Therefore, many studies have been made towards such a structure to represent 3D images due to its wide application in image segmentation, pattern matching, and other image processing algorithms. Both of these applications require an effective representation of image structure so that partition and merge can be easily operated [21–23]. In these types of application, different characters of target object need to be extracted in order to do further process. For segmentation algorithms, objects with same features, such as shape, size, color, and topological feature, need to be merged. Moreover, if images are stored in a form of boundaries and adjacency relations, some image processing algorithms can be done even without restoring the original image [24, 25]. Analysis of region structure is an important step before any 3D algorithms for computer vision. Therefore, development of an efficient and fast algorithm to extract topological structure is very crucial for image processing study.

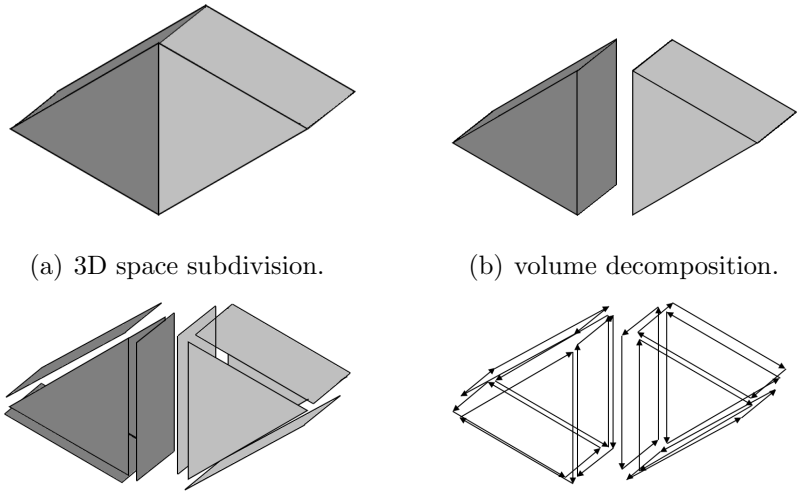
In the early 1990's, researchers studied many different approaches to represent 2D images using the topological structure. The most famous example is the Region Adjacency Graph (RAG) [26]. RAG describes images by vertex and edges that connect their neighboring region. Using this graph, it is really easy to find adjacent region for any given region using edges. However the major deficiency that RAG encountered is that RAG does not distinguish inclusion (there is no difference between adjacency and inclusion) and multiple adjacency relations [27, 28]. To solve this issue, W.G. Kropatsch proposed a dual-graph structure that consists of two multi-graph describ-

ing inclusion relation in 1994 [27, 28]. Addition to RAG, the pair of dual graphs has self-loops and multi-edge to indicate multiple adjacency relations. Dual graph solves the drawback of RAG, which trades increased space and time for adjacency information. As the result, any operation applied to a dual graph must be implemented twice, one for the prime graph and one for the dual graph. Dual graph structure is an excellent topological representation of an image. The idea can be easily extend to a higher dimension. Numerous research papers focus on this topic. Several studies based on 2D topological maps [29–32] proposed optimize methodologies to their application. Some of the models are also extend to 3D version [33,34]. The drawback of a dual graph is that it lack the explicit encoding of edges around vertices as existed in a combinatorial map.

In order to study the topological structure of a 3D model, researchers focus on Reeb graph, which represents the oriented structure of a 3D object. The Reeb graph describes the shape of the object but not its adjacency relationship [29]. Therefore, Reeb graph is not used in segmentation algorithm. Another approach is to use a combinatorial map [24, 35] or oriented boundary graph (OBG) [36, 37] to describe the topology structure of the image. The combinatorial map encodes all the inter-voxel elements and all adjacency information including regions Euler characteristic and Betti number using the basic element called dart. There are two ways to represent a combinatorial map. One representation is that using numbered segments to represent adjacent vertex and another representation is that using arrows instead. The oriented boundary graph uses nodes in the graph to represent regions of partition and every oriented edge in the graph to represent the corresponding surfaces. OBG uses surface elements to represent surface adjacency relations. Compare to a combinatorial map, OBG has its advantages like simple extraction and low memory consumption. However, it does not describe the topological character of a region. In contrast, A combinatorial map is a useful model for preserving topological characters and describing space subdivision in a 3D image.

3.1 Combinatorial Map Definition

A combinatorial map is an intervoxel elements representation of n-D images, which is a mathematical model describing a subdivided object as a set of elements from vertices (0-Dimension elements denoted as 0-cell), edges (1-Dimension elements denoted as 1-cell), and faces (2-Dimension elements denoted as 2-cell). This map includes the adjacency relations among different cells that describe the topological information of the images. Two i-cells are defined as adjacent when they share a common (i-1)-cell. Figure 3.1 illustrates an example of decomposition of a 3D object to extract combinatorial map. The combinatorial map can be extracted by sequentially decomposing the volume of the object, faces of the volumes and edges of each face. In order to represent geometry information of n-cells, adjacent relation is encoded and denoted as β. Darts are represented as arrows and they are the basic elements to compose face, edge, and vertex as shown in Figure 3.1(d).



(a) 3D space subdivision. (b) volume decomposition.

(c) face decomposition. (d) edge decomposition.

Fig. 3.1.: Elements Decomposition of a 3D Object to Extract Combinatorial Map.

The initial definition of combinatorial map in N-dimension is given in [38]. Its definition is further extended to allow represent objects with boundaries. Based on the definition given in [24], a n-dimensional combinatorial map is a n-tuple $M =$

$(D, \beta_1, \beta_2, ...\beta_n)$. β_n describes the relation between two i- dimension cells, where:

1. D is a finite set of darts;
2. β_1 is a partial permutation on D;
3. $\forall i 2 \leq i \leq n, \beta_i$ is a partial involution on D;
4. $\forall i 0 \leq i \leq n-2, \forall j 0 \leq i \leq i-2$, $\beta_i \circ \beta_i$ is a partial involution on D.

β_n is called n-sewn operation and this encoding is used to find adjacent region in combinatorial map. Two darts d_1 and d_2 are i-sewn if they satisfy either $\beta_i(d_1) = d_2$ or $\beta_i(d_1) = d_2$. β_1 represents relationship between two darts that share same face and same volume, an example of 1-sewn relation is shown as the the pair of two blue darts in Figure 3.2. The permutation of darts consists an orbit, as the four blue darts presented in Figure 3.2. And β_0 represents the reverse operation of β_1. If $\beta_1(d_1) = d_2$, then $\beta_0(d_2) = d_1$. β_2 represents relationship between two darts that share same edge and same volume. This operation is used to find adjacent dart. The pair of two red darts in Figure 3.2 are in 2-sewn relationship. β_3 represents relationship between two darts that share same edge and same face. Traversing between the faces in the same region are through the β_1 and β_3 operation. The pair of two green darts are in 3-sewn relationship. In combinatorial map, vertex (0-cell) can be represented as$< \beta_{21}, \beta_{31} > (d)$, edge (1-cell) can be represented as $< \beta_2, \beta_3 > (d)$, face (2-cell) can be represented as $< \beta_1, \beta_3 > (d)$.

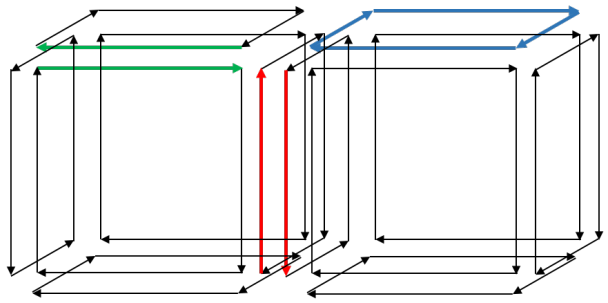
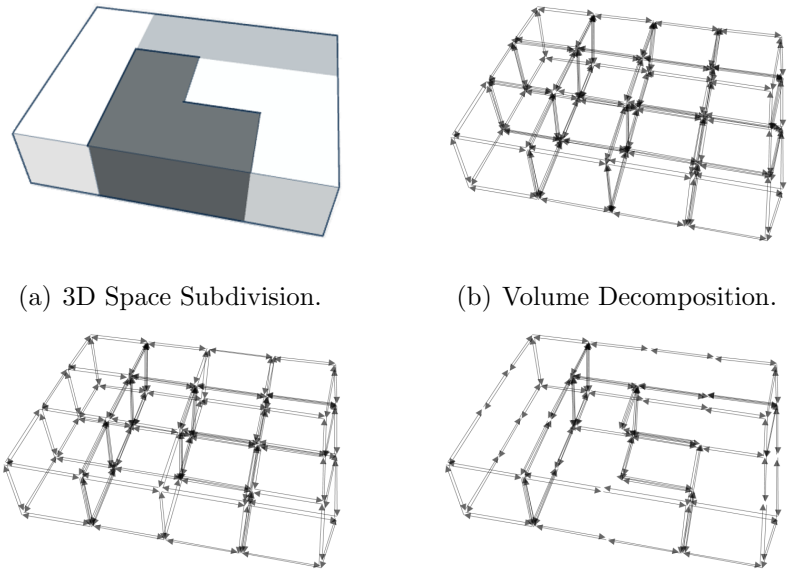


Fig. 3.2.: Darts in i-sewn Relationship

3.2 Combinatorial Map Construction

3.2.1 Complete Map (Level 0 map)

The complete map is the starting point of combinatorial map extraction. This map includes all the intervoxel elements of a given image. For a $n_1 \times n_2 \times n_3$ labeled image, its combinatorial map contains $n_1 \times n_2 \times n_3$ dart cubes corresponding to every voxel in the image. In Figure 3.3, Sub-figure (a) is a sample labeled image and Sub-figure (b) represents its corresponding level 0 complete map. In Figure 3.3(b), every voxel consists 24 darts, which results in 288 darts for a 4×3 image. Based on the definition of combinatorial map, the level 0 map is surrounded by an enclosing darts that describe its finite region. However, Figure 3.3(b) do not contain the finite region darts in order to make the map clear and understandable.



(a) 3D Space Subdivision. (b) Volume Decomposition.

(c) Face Decomposition. (d) Edge Decomposition.

Fig. 3.3.: Elements Decomposition of A 3D Object to Extract Combinatorial Map.

3.2.2 Level 1 Map and Face Removal Operation

Level 1 map is extracted from level 0 map by removing face darts between two adjacency voxels that have the same value or similar property. The map excludes the internal surfels information of each region of a 3D image using face removal operation that presented in Algorithm 1 [24]. In order to extract level map 1 from level 0 map, the image is scanned and faces that need to be merged are removed. Face removal is processed by removing the 8 darts between the two voxels. Figure 3.3(c) shows level 1 map obtained from image the given in Figure 3.3(a). Algorithm 1 in next page gives the procedure of removing faces from a given 3D image. The algorithm takes a 3D image and generates a level 1 map. For each single face, it removes the permutation of dart d and orbit of $\beta_2(d)$.

3.2.3 Inclusion Tree

After face removal operation of a 3D combinatorial map, the generated level 1 map may lose volume connection information if the map contains two or more regions that one of the regions is surrounded by another region [38]. This kind of disconnection is called volume disconnection. As the result, the combinatorial map will lose the topological information between these two regions. Therefore, a supplementary data structure is required to save connection information between different regions as shown in Figure 3.4(b).

An inclusion tree is designed to solve this situation. As shown in Figure 3.4, the inclusion tree on the right side includes the connection information of the image on the left. The root of the tree contains the boundary of stack image. Each node under the tree root represents a region corresponding to the regions in Figure 3.4(a). Different regions that share the same parent are saved as the child nodes of that parent node. Nodes in the tree contain boundary information or darts of a corresponding region. Algorithm 2 in the next page gives the procedure to generate inclusion tree structure.

Algorithm 1 Level 1 Map Extraction

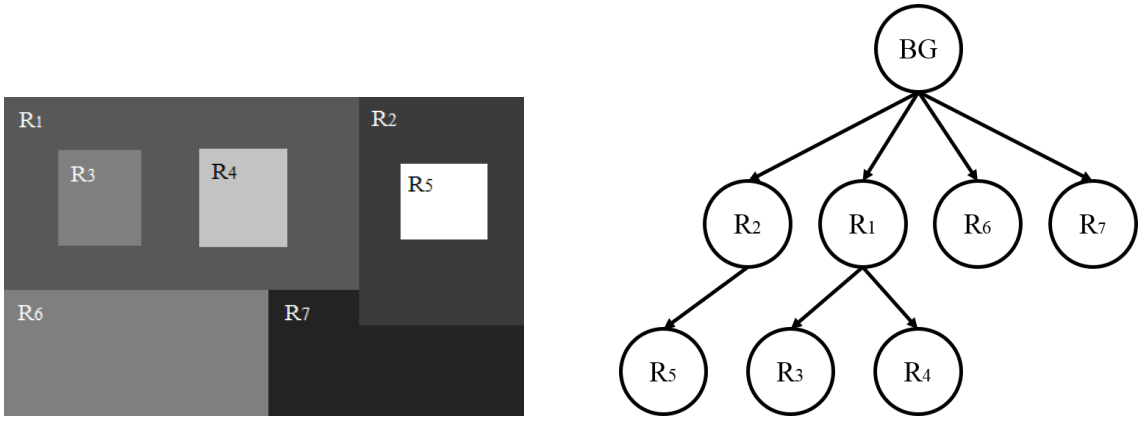
Require: level 0 map M with dimension $(n_1 * n_2 * n_3)$

```
for k = 0; k < n1; k++ do
  for j = 0; j < n2; j++ do
    for i = 0; i < n3; i++ do
      if M(i, j, k) == M(i+1, j, k) then
        remove Orbit(d) and Orbit(β2(d))
      end if
      if M(i, j, k) == M(i, j+1, k) then
        remove Orbit(d) and Orbit(β2(d))
      end if
      if M(i, j, k) == M(i, j, k+1) then
        remove Orbit(d) and Orbit(β2(d))
      end if
    end for
  end for
end for
return level 1 map M
```

An inclusion tree is used to identify the region that a particular voxel belongs to. A tree traversal algorithm gives the procedure to search a region in the tree structure. The algorithm iteratively searches the tree level by level to identify a specific region.

3.2.4 Level 2 Map and Edge Removal Operation

Level 1 map represents the boundary information about the input 3D image. However, it still contains too much redundant information (darts), and is not efficient enough to describe the topological relationship of the image. To further minimize the map, adjacent faces that belongs to the same region are merged by removing edge



(a) Original Image. (b) Inclusion Tree.

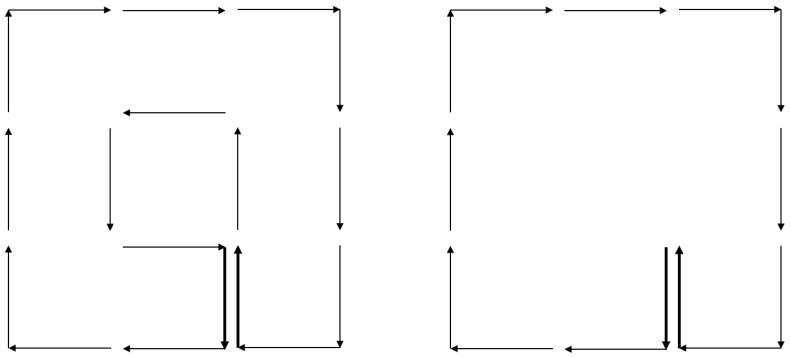
Fig. 3.4.: 2D Image with Inclusion Tree

Algorithm 2 Inclusion Tree Generation

Require: Background Node R_{BG}, Region List L

for every R in region list L **do**
  Node $R_c = R_{BG}$
  while 1 **do**
    for every subnode R_s in R_c **do**
      if R belongs to R_s **then**
        $R_c = R_s$
        continue
      end if
    end for
    insert R as subnode of R_c
    break
  end while
end for
return R_{BG} with Region Stored as SubNode

between them. To obtain the level 2 map, the whole images are scanned slice by slice and any edges that their degrees (the number of times of each distinct faces incident to this edge) are equal to two are also removed. During removal, some criteria must be applied in order to avoid removal of fictive edges.



(a) Fictive Edge. (b) Regular Edge

Fig. 3.5.: Fictive Edge Example

A fictive edge is an edge that its removal will result in disconnection of two faces. In Figure 3.5(a), the bold dart pair indicates a fictive edge. If this edge is removed, the relationship between two faces will be lost. In Figure 3.5(b), the bold dart indicates real edge, edges like this should be removed. Earlier study has presented Algorithm 3 [24] to determine if the edge has to be removed as shown below.

3.2.5 Level 3 Map and Vertex Removal Operation

The last construction of vertex removal operation is to removal every vertex that has two edges directly connect to it. Then for each dart exist on the map, the vertex removal operation trace through it along its direction and check if the degree of the vertex is equal to two, if true, the vertex will be removed. Figure 3.6(d) shows the corresponding level 3 map of Figure 3.6(a).

To construct the complete combinatorial map, Algorithm 4 provides a complete

Algorithm 3 Edge Removal

Require: level 1 map M with dimension $(n_1 * n_2 * n_3)$

for every dart d in level 1 map M **do**

if $\beta_{23}(d) \neq \beta_{32}(d)$ **then**

return;

end if

if $\beta_0(d) \neq \beta_2(d) and \beta_1(d) \neq \beta_2(d)$ **then**

return;

end if

$d_0 = \beta_d; d_1 = \beta_{21}(d); d_2 = \beta_{20}(d); d_3 = \beta_1(d);$

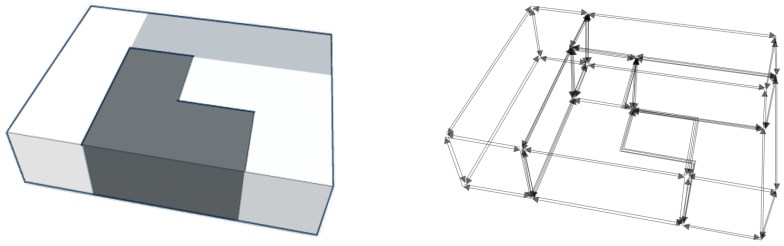
$1 - sew(d_0, d_1); 1 - sew(\beta_3(d_1), \beta_3(d_0));$

$1 - sew(d_2, d_3); 1 - sew(\beta_3(d_2), \beta_3(d_3));$

$Removed, \beta_2(d), \beta_3(d), \beta_{23}(d),$

end for

return level 2 map M



(a) Original Image. (b) Combinatorial Map

Fig. 3.6.: Minimal Combinatorial Map Generation

procedure that uses the cell operation proposed in previous sections. The algorithm takes labeled image and constructs combinatorial map level by level.

Algorithm 4 Combinatorial Map Extraction

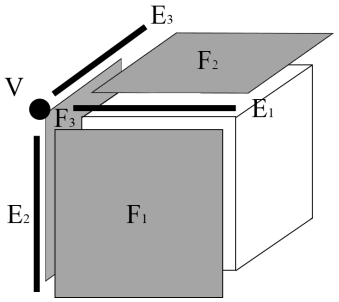
Require: level 0 map M with dimension $(n_1 * n_2 * n_3)$

```
for k = 0; k < n1; k++ do
  for j = 0; j < n2; j++ do
    for i = 0; i < n3; i++ do
      if merge condition for f1(resp. f2, f3) then
        faceremoval(f1(resp. f2, f3))
      end if
      if local degree of e1(resp. e2, e3) = 2 then
        if e1(resp. e2, e3) is not fictive edge then
          edgeremoval (e1(resp. e2, e3))
        end if
      end if
      if local degree of v =2 then
        vertexremoval(v)
      end if
    end for
  end for
end for
return level 1 map M
```

3.2.6 Encoding Method

A combinatorial map represents geometry information and boundary information of a labeled image. Different encoding methods are applied to the map depending on their applications. If only the geometry information of a given region is necessary but do not care about a voxel that belongs to a specific region, it is only necessary to use a hierarchical model. In this model, faces are represented as numbered segments and edge are represented as oriented darts [39]. If an application requires both

information, [24] presents another solution to encode the combinatorial map. It requires a 3D matrix to represent all the intervoxel elements of a 3D labeled image, and uses 7 bits to represent the intervoxel elements between voxels as presented in Figure 3.7. For each element in a intervoxel matrix, it contains 3 bits for the front face, the top face, and the left face, 3 bits for the top front edge, the front left edge and the top left edge, 1 bit for the top left vertex. Figure 3.8 shows the intervoxel representation for one voxel. To represent the whole boundary for 3D image, it requires $(n_1+1)\times(n_2+1)\times(n_3+1)$ matrix to represent a $n_1\times n_2\times n_3$ image. [24] also proposed the link between the darts of a topological map with triple (vertex, edge, face). Triple $(f_1.(i,j,k), e_1.(i,j,k), v.(i,j,k))$ is equal to d_0 at voxel (i,j,k).



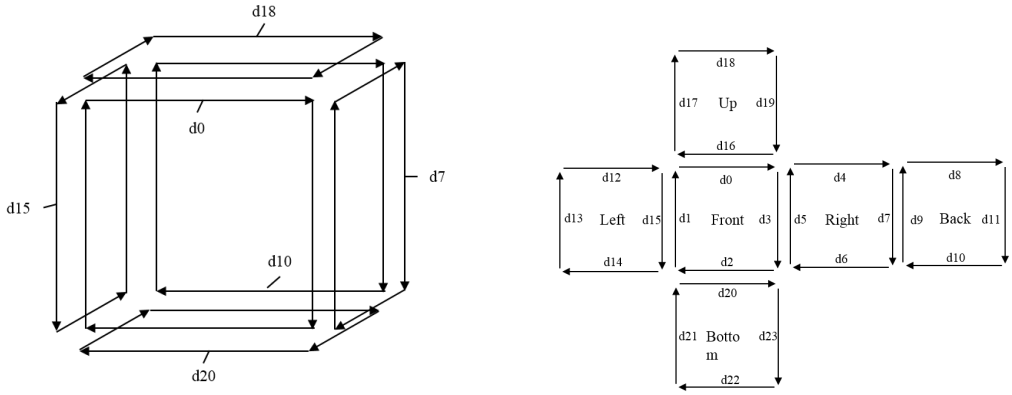
(a) Boundary elements.

Bits	7	6	5	4
Cell	F3	F2	F1	E3
Bits	3	2	1	0
Cell	E2	E1	V	

(b) Bit Representation

Fig. 3.7.: Combinatorial Map Encoding Matrix

An intervoxel matrix encoding method requires more memory space than the hierarchical model. But using intervoxel matrix gives user more flexibility to remove intervoxel elements (by clearing the corresponding bits in the intervoxel matrix) and retrieve voxel location. Algorithm 4 proposed in [24] gives a sequential method to extract a combinatorial map of a labeled 3D image. Simple removal and retrieve operation make the matrix very suitable for 3D segmentation application. It allows user to select a specific region in segmented 3D image stack to get information like volume and shape of that region.



(a) Dart Number. (b) Extend Dart Map.

Fig. 3.8.: Dart Number Notation.

The main drawback of an intervoxel matrix is that it cannot preserve fictive edge. This map loses face connection information during edge removal in level 2 map construction (this problem is discussed in section 3.2.4). Different encoding method can be applied to address this problem. Instead of using 7 bits to represent boundary information, and retrieve dart d based on a triple (vertex, edge, face), every dart and its direction (used to represent i-sewn operation) are encoded into the matrix. Using this encoding, 3 bits are used to represent each dart as shown in Figure 3.8(b). Using 1 bit to represent its existence on the map and 2 bits to represent its direction. During initialization of the map, existence bit are set to 1 to represent the darts belongs to the map. To remove a dart, the existence bit is set to 0 and its corresponding direction bits are set to 3 (other direction values represent the vertex between two darts is removed) to represent its removal. It is certainly guaranteed that the dart cannot point to direction 4 because it is removed in level 2 map construction, as shown in Figure 3.3(b). Direction value equals to 0 indicates dart points to corresponding location 1 as shown in Figure 3.9; Direction value equals to 1 indicates dart points to corresponding location 2 and so on. Encoding direction value allows us to track and keep the fictive edges.

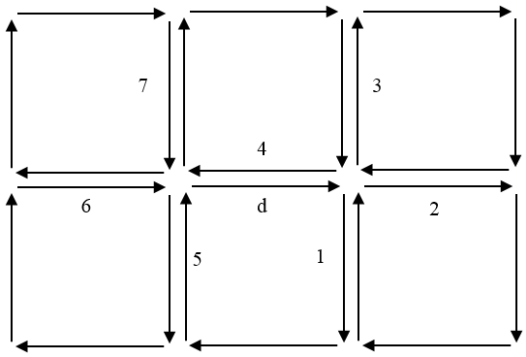


Fig. 3.9.: 1-Sewn Direction

To perform i-sewn operation $(\beta_1, \beta_2, , \beta_i)$ on a dart, a look-up table is used to speed up the processing. The tables in Appendix I includes the encoded information to find different i-sewn operations for a given dart. In Table A.1, the first column shows the dart location corresponding to a voxel (as illustrated in Figure 3.8, the darts corresponding to a voxel is assigned with different index to identify them. Figure 3.8(b) is the flat shape of the cube of the voxel in Figure 3.8(a)). The 2 bits direction number of a dart is used to find its 1-sewn operation using column 2 to 5. For example, if the current number of dart d is d_{10} corresponding to a voxel at (i, j, k) and its direction number is 1. Consequently, its 1-sewn operation $\beta_1(d) = d_{10}$ corresponding to voxel at $(i + 1, j, k)$ are obtained through the table.

Table A.3 in Appendix I represents the encoded information to find 2-sewn operation for a dart. For instance,to look for $\beta_2(d_7)$ at voxel (i, j, k), first, testing if voxel at (i, j, k) is equal to voxel at $(i, j, k + 1)$ is applied. If they are equal, then the $\beta_2(d_7) = d_5$ at voxel $(i, j, k + 1)$. otherwise $\beta_2(d_7) = d_9$ at voxel (i, j, k).

Table A.4 in Appendix I represents the encoded information to find 3-sewn operation for a dart. β_3 indicates the darts that share the same edge and the same face. Therefore, look up is very straight forward. 3-sewn operation of dart d_{12} at voxel (i, j, k) is d_4 at voxel at $(i + 1, j, k)$.

3.3 3D Segmentation with Combinatorial Map

Many papers proposed very interesting results about region-based image segmentation. Region merge based 3D segmentation algorithm usually uses local region variation and adjacency information as merge criterion. [40, 41] describe a region growing and spectral clustering segmentation algorithm. They treated each pixel in the image as the initial region and merge regions based on a dissimilarity criterion repeatedly until the number of regions is less than a preset value. In [42], it detailed discussed the topological map that commonly used in region-based segmentation including region adjacency graph, discrete map, and combinatorial map, and summed up several different segmentation algorithms based on the topological map. [36, 43] further extended this model to apply on 3D image and proposed split and merge algorithm on topological maps. [22, 44] proposed 3D image segmentation algorithms based on the variants of the 3D topological map. All the results indicate the segmentation algorithms will benefit from the information stored in the topological map.

Compare to other algorithms, [23] provides excellent segmentation result using local variation as criterion and RAG in merging two regions to produce a homogeneous region. Due to its significant results, this method is further developed using a hierarchical combinatorial map that represents a 2D image in 2D segmentation algorithm [45] and 3D segmentation algorithm using 3D combinatorial map [22]. [22] presents similar segmentation method in 3D image as result in [23] and it provides very promising processing time to segment 3D medical images. It states that it takes 25.10s to segment a 3D image with dimension $256 \times 256 \times 111$ and merge 431486 initial regions to 30179 remaining regions. It gives same good result as the method proposed in [23]. Even though the processing time is acceptable to segment a 3D image, it is still not fast enough for real-time application. In this thesis, an intervoxel matrix representation to encode combinatorial map as discussed in section 3.2.6 [24] are used. This encoding method is suitable for parallel algorithm in GPGPU implementation.

Therefore, processing time can be improved and the algorithm can be applied in real time application such as Voxx 3. In this section, how to use combinatorial map in segmentation algorithms and the encoding methods in combinatorial map representation are discussed.

3.3.1 Segmentation Algorithm

The segmentation algorithm uses a local criterion proposed in [22] and the encoding method proposed in [24]. Since face disconnection does not influence the segmentation result, existence of fictive edge in combinatorial map can be ignored during segmentation. Thus, the algorithm uses 7 bits intervoxel matrix. In the algorithm, the intensity difference between two adjacency pixels called external variation(denoted ExtVar) is used as the merging criterion. For any two neighboring regions, the external variation(denoted ExtVar (R1,R2)) is the minimum intensity difference between the pixels aside the border of the two regions. Each region in the image also contains an internal variation (denoted IntVar). The internal variation is the maximum intensity difference between all the voxels in the region. The interval variation of the new region after merging has been approved equal to the external variation of two regions before merging [22].

Then two regions are merged into a single region based on the criterion given in equation 3.1. The external variation between the regions should be smaller than their minimum internal variations [23]. The equation is given below:

$$ExtVar(R_1, R_2) \leq min(IntVar(R_1), IntVar(R_2)) \tag{3.1}$$

To improve the tolerance to the internal variance of a region, a threshold function is added to the internal variations of the two candidate regions, which allows to merge them when their external variance larger than their internal variance [23]:

$$ExtVar(R_1, R_2) \leq min(IntVar(R_1) + k/size(R), IntVar(R_2) + k/size(R)) \tag{3.2}$$

In the equation, k is a constant defined by the user. Size(R) represent the number of voxels are in the region R. Higher k value represents larger internal variation that the region can tolerance. In contract, lower k value denotes the region can only hold a smaller internal variation. As shown in equation 3.1, this value also controls the maximum volume of a region, since the variation tolerance (k/size(R)) will decrease as the region size increase.

Algorithm 5 in the next page provided a detailed procedure of this 3D segmentation algorithm. At the beginning, a queue that contains every boundary in the image is created for further processing. In our case, initially, boundary is the every face element in the combinatorial map. These faces in the queue are sorted based on its external variation. Faces with low external variation are processed first, and those with high external variation is processed later. After all the faces are stored in the queue, the algorithm scan each face in the queue and find the corresponding region on the different side of that boundary. Faces with the same region are removed in the combinatorial map. Otherwise, a test is applied to check if these two regions meet the merging criterion by comparing the external variation of the border with the regions internal variations(as shown in equation 3.1). If the external variance is smaller than the two regions internal variance, then two regions are merged by remove the boundary in the combinatorial map.

3.4 Result

To analysis the running time of the 3D segmentation algorithm, the dimension of the input image are assumed as $n_1 \times n_2 \times n_3$ corresponding to row, column, and slice. Running time of the algorithm can be partition into two parts. At the beginning, the algorithm initializes the list of faces based on each face external variant. Every face inside the image are inspected to calculate the external variant of each face. There are total $3 \times n_1 \times n_2 \times n_3 + n_1 \times n_2 + n_1 \times n_3 + n_2 \times n_3$ faces in a $n_1 \times n_2 \times n_3$ stack

Algorithm 5 combinatorial map based 3D segmentation algorithm

Require: level 0 combinatorial map M, threshold constant k

```
Q ← sort all faces in M based on it external variation
while !Q.empty() do
  F = Q.pop()
  R_1 = region(F); R_2 = region(β_3(F))
  if R_1 != R_2 then
    if ExtVar(R_1, R_2) ≤ min(IntVar(R_1)+k/size(R), IntVar(R_2)+k/size(R))
    then
      R_1 = R_1 + R_2
      remove(R_2)
      IntVar(R_1) = ExtVar(F)
    else
      continue
    end if
  end if
  remove(F)
  remove(E adjacent to F)
end while
return segmented combinatorial map M
```

image, the running time complexity of this operation is O($n_1 \times n_2 \times n_3$). Then each faces are sorted into the queue from low to high based on its external variation. In this step, count sort is used due to its linear run time. In this application, count sort is faster than quick-sort practically. Depend on different image type, the maximum external different is 255 for 8 bits image, 4095 for 12 bits image and 65535 for 16 bits image. Counting sort running time complicity is O($n_1 \times n_2 \times n_3$).

As shown in the second part, the main loop checks through each face inside the list. To analysis the time complicity of the second part, each step inside the loop has to be analyzed. In the program, a hash table is used to establish the relationship between voxel location and region. Each region contains every voxel inside that region and the internal variation. Theoretically, it takes O(1) to extract region R_1 and R_2 and judge if two regions are equal. It is difficult to calculate time complicity of region merge locally, and it is easier to calculate globally. In the worst case, all of the voxel in the image belong to the same region. It will take O($n_1 \times n_2 \times n_3$) to perform merge operation. To find the external variance between two regions, the algorithm traces the surface until an edge is meet. In this step, the maximum size of a face is assumed to be $(n_1 \times n_2 + n_2 \times n_3 + n_3 \times n_1)/3$. Using depth first search or breadth first search will take O($n_1 \times n_2 + n_2 \times n_3 + n_3 \times n_1$) to perform one search. In the last step, remove faces and edges takes O(1). Sum all the time complicity together, total time complicity of combinatorial map based 3D segmentation is O($(n_1 \times n_2 + n_2 \times n_3 + n_3 \times n_1) \times n_1 \times n_2 \times n_3$). The time complicity of algorithm is analyzed based on monochrome images. For color images, the algorithm can be applied to each channels or preform once by giving each channel a different weight. The time complicity is the same as gray-scale image.

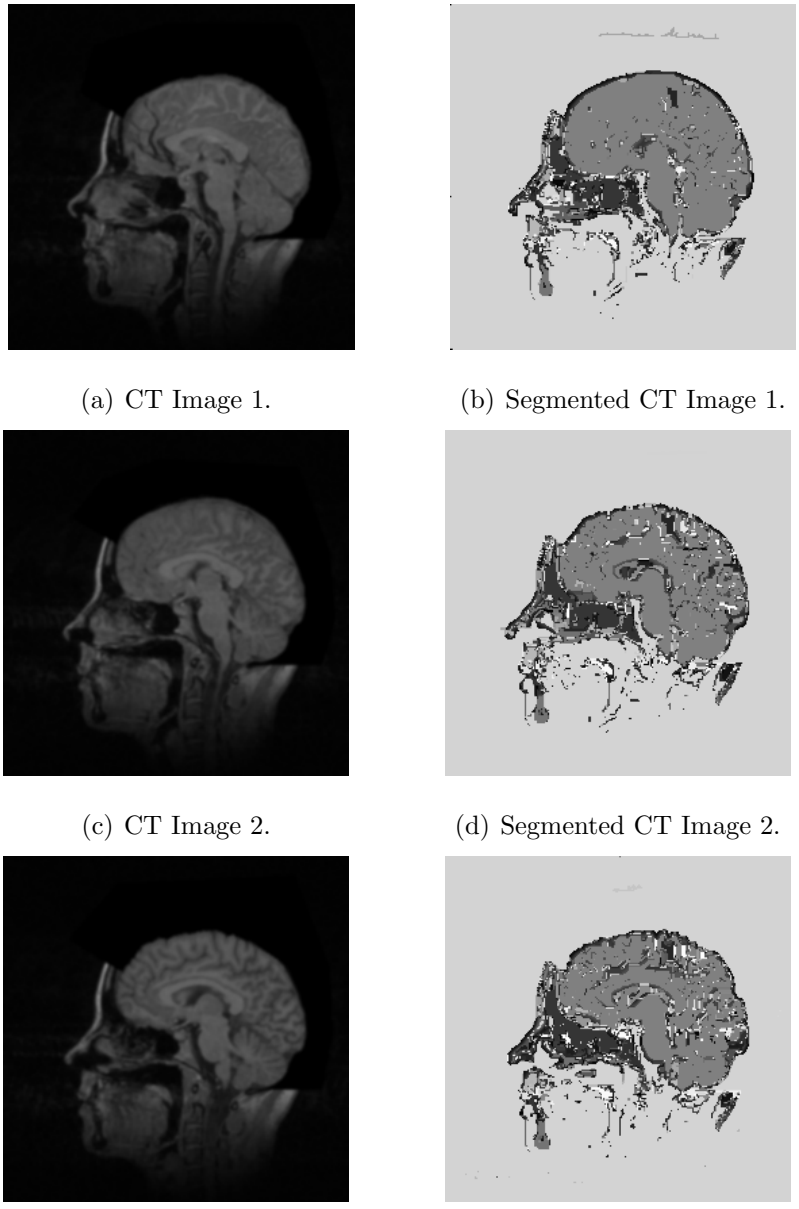
Table 3.1 includes image information and processing time for the five steps of the segmentation algorithms. Image is not pre-processed which means the number of initial regions is equal to $n_1 \times n_2 \times n_3$. The fourth row in the table is final region numbers. As concluded in the table, running time of faces initialization and region merge is relatively small compare to other processes. In the source code, the hash table implemented in C++ STD library is used. So the fact that the running time of region fetch is not discussed here. The most time-consuming procedure is region fetch and face edge removal. In this two step, a breadth first search algorithm is used to trace the surface and the time consumed by this two step is worse than linear.

Table 3.1: Running Time Table of Combinatorial Map Based Image Segmentation

Image	Stack 1	Stack 2	Stack 3	Stack 4
Dimension	128×128×5	256×256×5	256×256×10	512×512×10
Threshold	50	50	500	500
Final Regions	3766	15779	39669	75185
faces Initialization	0.074s	0.138s	0.341s	1.2s
region fetch	2.43s	6.76s	10.85s	68s
EXTVAR Calculation	1.73s	5.32s	9.03s	50s
Region Merge	0.84s	2.69s	4.19s	26s
Face Edge Removal	2.66s	6.93s	11.67s	70s
Total Run Time	7.73s	21.8s	36.08s	214s

Figure 3.10 and Figure 3.11 present segmentation result obtained using different datasets. Figures on the left side are input images and figures on the right side are segmented images, voxels in segmented images with same color belongs to same regions. In Figure 3.10 CT segmented results, the background is merged into one region and most of the brain are merged into same region. The mouth and the neck are also merged into the background. This is because the of voxel difference between these regions are closer to background than with other region.

In Figure 3.11(b, e, h) and Figure 3.11(c, f, i), most of the cells in the image are segmented into different regions. However, cell boundaries are segmented into different regions due to the boundary voxels have small internal variant and external variant are relative large compare to its adjacent regions. Figure 3.11(b, e, h) are segmented results with merging constant k = 500, blood vessel are connected in original Image but segmented into different parts. Figure 3.11(c, f, i) are segmented results with merging constant k = 2000. As shown, blood vessel are segmented into the same region.



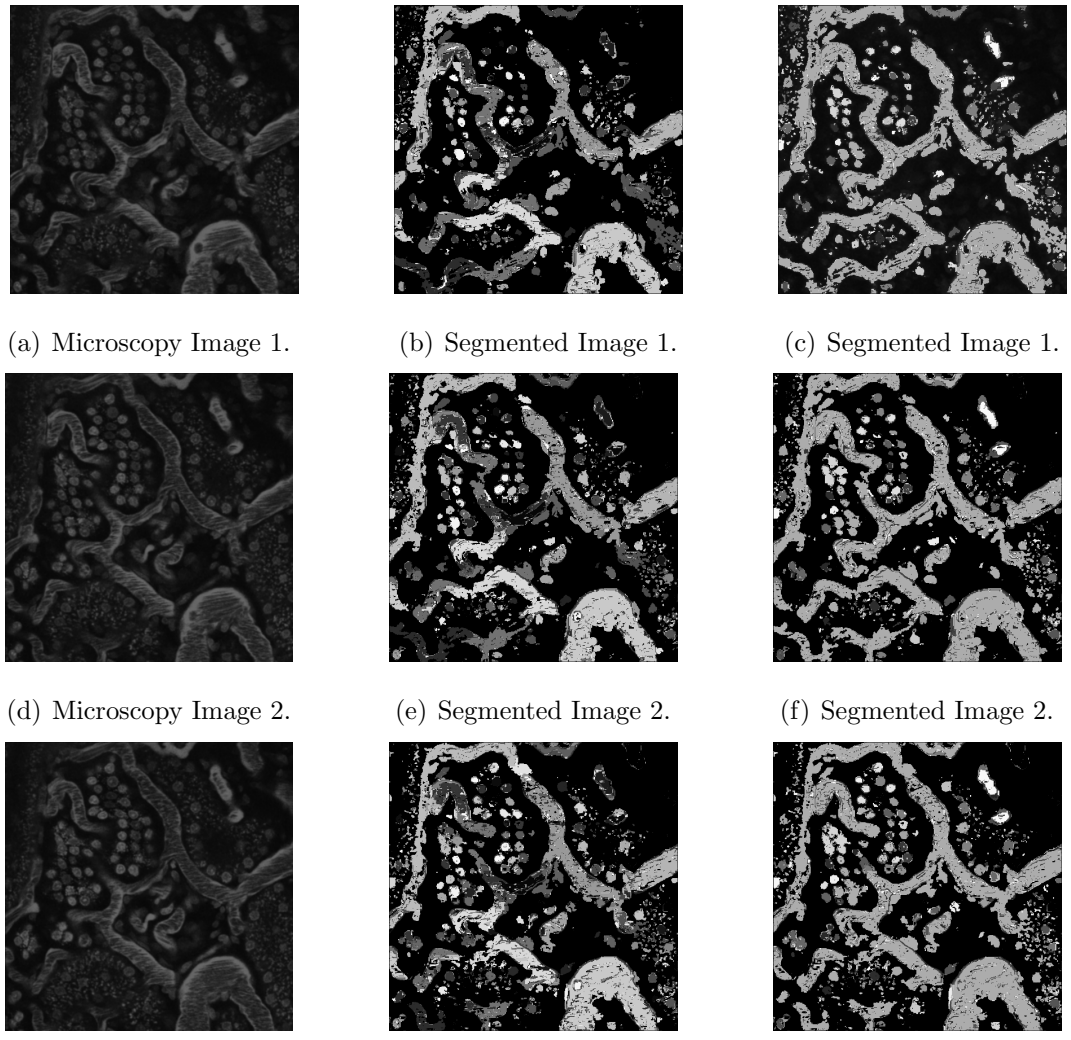
(a) CT Image 1. (b) Segmented CT Image 1.

(c) CT Image 2. (d) Segmented CT Image 2.

(e) CT Image 3. (f) Segmented CT Image 3.

Fig. 3.10.: Three Consecutive CT Image with Segmentation Result.

Figure 3.11(a) is segment result with background(maximum region) set to 0. Figure 3.11(b) is segmented result with background set to 0 and region that volume smaller than 50 are removed.

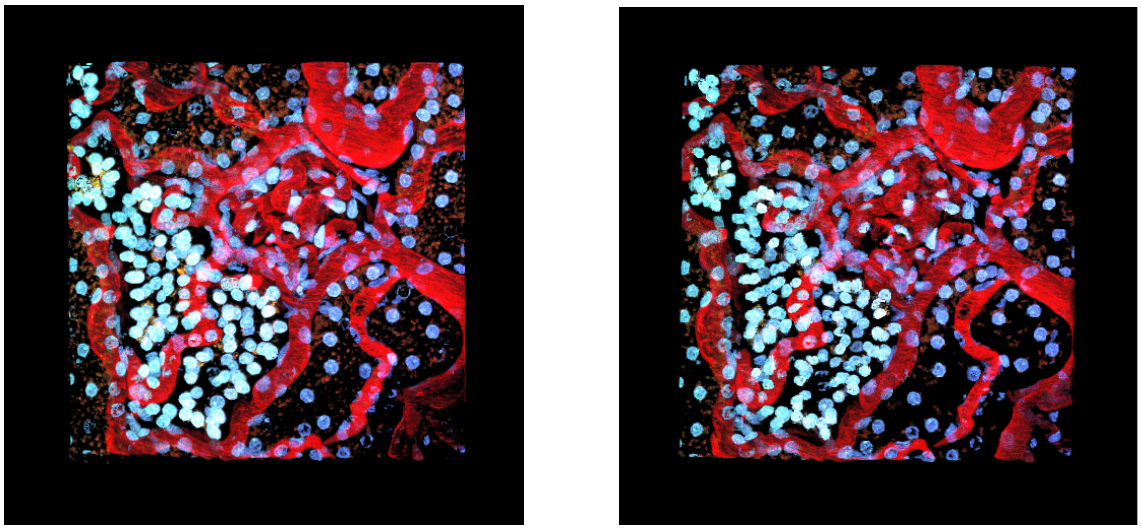


(a) Microscopy Image 1. (b) Segmented Image 1. (c) Segmented Image 1.

(d) Microscopy Image 2. (e) Segmented Image 2. (f) Segmented Image 2.

(g) Microscopy Image 3. (h) Segmented Image 3. (i) Segmented Image 3.

Fig. 3.11.: Three Consecutive Microscopy Image with Segmentation Result.



(a) Segmented Image with Merging Constant k = 500.

(b) Segmented Image with Small Region Removed.

Fig. 3.12.: Segmented Microscopy Image DataSet.

4. GPGPU AND NVIDIA FERMI ARCHITECTURE

GPGPU is the abbreviation of General-purpose computing on graphics processing unit (GPU). The core concept of GPGPU is performing computation on graphics processing unit(GPU) instead of using central processing unit(CPU), due to GPU is more capable of handling a large amount of data when same computation process is applied on data. However, the existence of GPU is not intended to replace CPU. Since both of them have significantly different designs and suitable for different tasks. While CPU is designed for fast processing on a per-core basis, it handles complex operations on a few stream of data and a high percentage of conditional branches. On the contrary, GPU design aims to take a large amount of data and performing the same operations very quickly.

Due to the advancement of GPU hardware, image processing on GPU is regarded as a promising and efficient way. Initially, graphical hardware was used to accelerate to convolution [46]. Since convolution is perfectly suitable for single instruction multiple data (SIMD) operation, which means the result is calculated in the same way for each pixel, the convolution with kernel size 7 on 128 3D images improved from 2.0s on CPU to 0.48s on GPU and 21.3 s on CPU to 3.7s on GPU. Due to the significant result, other image processing operations like interpolation, pixel statistic histogram and Affine transformation are implemented using GPU. Image processing on GPU are further extended to many other aspect [47], such as image registration [48–51] and image segmentation [52]. Many image processing algorithms that are too complicated for real-time application using CPU are now become practical using GPU. By processing images on GPU, researchers can focus developing more complex and effective algorithms to improve results without considering tread off quality with processing time. Therefore, the 3D segmentation algorithm as discussed in section 3.3.1 can be

implemented on GPU kernel to make up the cost of processing time. This thesis focus on the combinatorial map operation and generation at this time. The implementation and optimization of the combinatorial map operation on GPU is provided. The next step of this work is to completely integrate 3D segmentation algorithm on GPU using the topological structure that proposed in this thesis.

Even though the GPU function is usually much faster than CPU function when processing a large amount of data, it is still necessary to optimize the GPU function to achieve better performance. To further improve GPU function performance, GPU kernel optimization technique is proposed and used in GPU programming. [53–55] proposed optimization principles to evaluate CUDA application performance and these principles are used in this thesis to guide the optimization technique. This chapter first discuss the GPU programming model, kernel function, and CUDA GPU architecture. Then kernel optimization technique for combinatorial map operation is proposed in the end of this chapter.

4.1 CUDA Programming Model

The classic GPGPU technology is built on 3D graphic accelerator application programming interface (API), which provides operation of GPU graph pipeline. The API allow users to plot vertex, perform vertex transformation, enable lighting effect, map texture on vertex and display computer generated graphics on screen [56]. Traditional general computation using GPU rendering API focus on mapping data to computer screen buffer and design fragment shader to process data. When performing general programming using graph pipeline API, data is presented as vertex or texture information, and the result is retrieved from the output of the graph pipeline. During computation, the operation, the processing sequence, and the priority level of thread are allocated by operation system. Therefore, using GPU graph pipeline to perform computation requires knowledge of graph pipeline and GPU architecture.

In nowadays, programming languages and platform like CUDA and OpenCL are used to perform general purpose computation on GPUs. This thesis focuses on Nvidia Fermi architecture and uses CUDA API to compute combinatorial map. In 2006, Nvidia introduced GeForce 8800 with CUDA, which is the first c-based development API. Compared to traditional GPGPU, CUDA provides more convenient and effective programming model in parallel processing, which allows programmers to perform threads operation. The importance of threads operation is significant. If users can optimize the activity of each thread based on the algorithm, the performance of the GPU program will be improved significantly. In CUDA, a hierarchy of thread groups that called warp is used to organize data. Users can wake a certain amount of threads based on the amount of data, and mapping them with input data. Therefore, every thread group can be scheduled on available the streaming processor in any order, concurrently or sequentially [56].

Using thread group is also the solution to solve the problems that raised by hardware diversity. It allows the same code to run on different GPUs, like GeForce, Quadro, and Tesla. For example, if a CUDA program contains N thread groups and it is running on GeForce 8400GT (which only contain two streaming processors as shown in Figure 4.1), every two thread groups will be arranged to a streaming multiprocessor(SM) sequentially and threads in each thread group will be processed concurrently. If the same function is executed on a GeForce GTX295 which contains 60 SMs, Then each group will be scheduled to 60 SM concurrently:

1. If $N < 60$, then there exists idle SMs.
2. If $N = 60$, then all the SM will compute at the same time.
3. If $N > 60$, then some thread groups are waiting in the queue, and serial computing occurs.

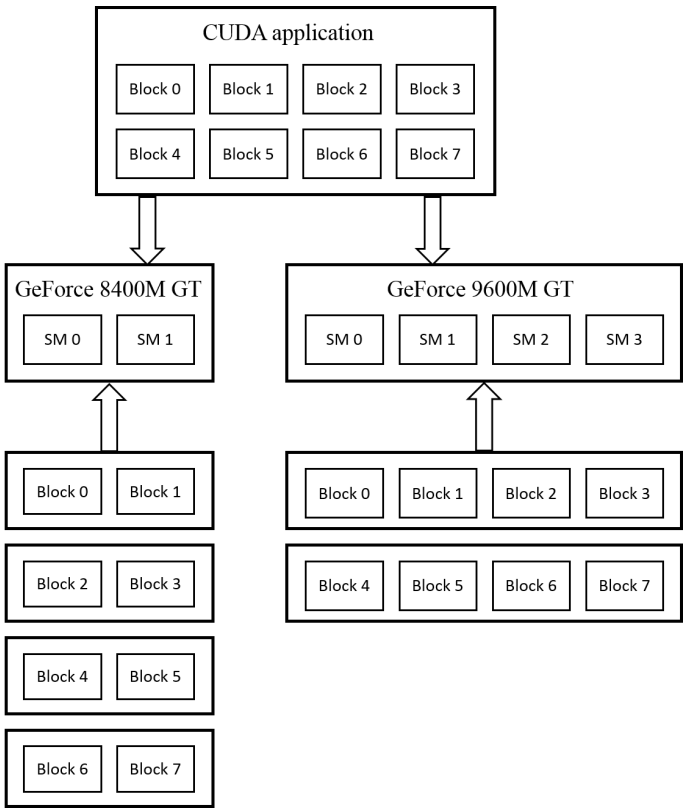


Fig. 4.1.: CUDA Programming Model

4.2 CUDA Kernel

The CUDA application is similar to the C/C++ application. Both of them consist of a return variable type, a function name and parameters. The most significant differences between them are that CUDA classifies functions into three types that based on the location where the application is executed and CUDA applications contain the thread group information followed by the function name. The equation 4.1 below provides the syntax of the CUDA kernel function.

$$[Qualifier][Return\ Type]Function\ Name <<< Block, Grid >>> (...Parameter...) \tag{4.1}$$

In C/C++ application, the functions are always executed on CPU. However, in CUDA application, the functions that called from CPU and executed on CPU is des-

ignated as host functions. The functions that called from CPU and executed on GPU is designated as global functions as shown in Figure 4.2. The functions that called from GPU and executed on GPU is designated as device functions. In Figure 4.2, the red qualifier before the return type indicates where the program is executed. In this example, the program vecAdd is called from CPU and executed on GPU. The grid and block qualifier followed by the kernel name (VecAdd) indicates the number of threads in a thread group and the number of thread group in an application. In this case, only 1 group of threads is activated and the dimension of this group is defined as N. The predefined identifier threadIdx allows users to manipulate threads to either access global memory or perform operations. In Figure 4.2, N awakened threads access integer array A and B and store summations in array C. Section 4.4 discuss how to calculate and optimize threads number in detail.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Fig. 4.2.: CUDA Kernel Function

4.3 GPU Architecture

This section focuses on the Nvidia GPU architecture, specifically the Fermi based GPU. Fermi is the name of a GPU architecture developed by Nvidia. The first Fermi based GPU featured up to 512 CUDA cores. These 512 cores are organized in a 16

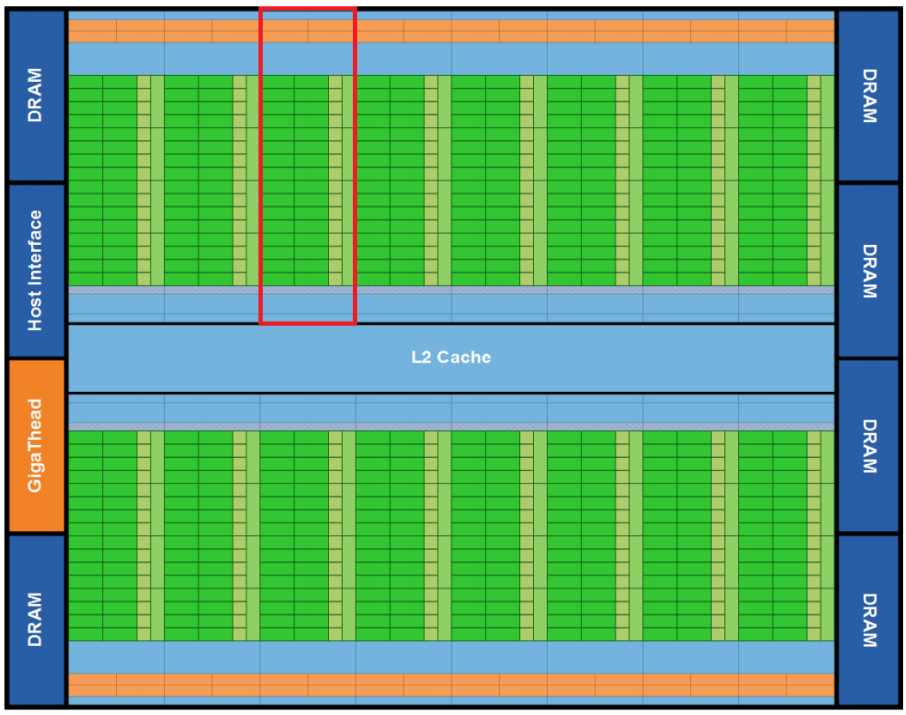


Fig. 4.3.: Nvidia Fermi Architecture (Source: NVidia)

stream processors of 32 CUDA cores in each (this is the reason that CUDA operate threads as a unit of 32 called warp). Figure 4.3 is a block diagram of the Fermi architecture [57]. In Figure 4.3, the red block represents a streaming processor (SM) and there are 16 SMs in total. As shown, every SMs are supported by L2 Cache, host interface, GigaThread scheduler and DRAM interfaces. Figure 4.4 is a detailed streaming processor block diagram [57]. Each Fermi SM consists of 32 cores, 16 load/ store units, 4 special function units, a 32k-word register file, 64K configurable RAM, and thread control logic.

CUDA memory also plays a significant role in Fermi architecture. CUDA API also gives user administration to control device memory, so the run-time library can allocate, deallocate and copy data in device memory. In CUDA, memory can be classified into three type:

1. Private local memory that can only be read and write by every thread.

2. Shared memory that can be read/write by every thread in each block.

3. Global memory and constant memory that can be accessed by all the threads in the CUDA kernel.

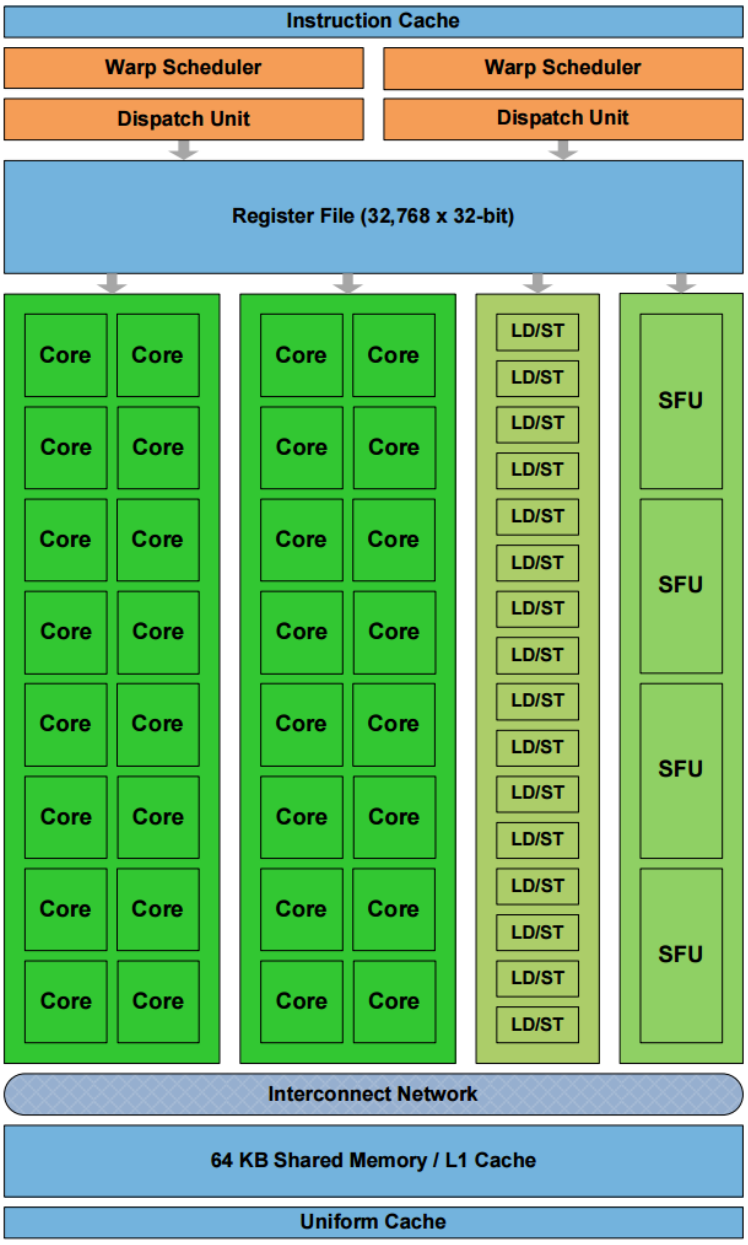


Fig. 4.4.: Fermi Streaming Multiprocessor (Source: NVidia)

Table 4.1: GPU Memory

Memory name	Hardware	Scope	Hardware location
Shared memory	On-chip memory	Block	Streaming Processor
Private local memory	Register	Thread	Streaming processor
Global memory	Device memory	Every thread	DRAM
Constant memory			

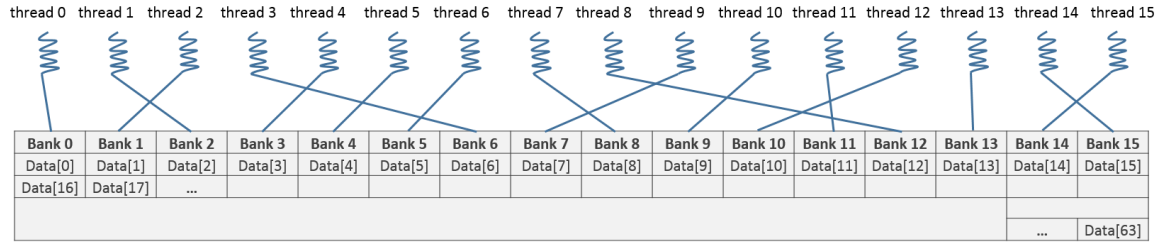
In GPU, private local memory is implemented using register. In typical situations, variables defined in the CUDA applications are stored in these registers. For example, if user define int i = 1 in the code, i is a variable that stored in register. As shown in Figure 4.4, registers are distributed in each SM, and provide private storage space for each thread. Using registers in CUDA applications allow fast data access. But registers can only provide a limited memory space. A GPU with computation capability (CC) 1.0 only contains 4KB registers, and a GPU with CC 2.0 contains 16KB registers. Usually, access latency of register is negligible. However, if developer uses more registers than the hardware has, overflow will occurs. Then, the overflowed data will be stored in global memory. Accessing these data will dramatically decrease the performance of the program. It is important to monitor the numbers of registers that allocated by a thread and the number of threads in each block. For instance, if assigning more threads in a block, then the number of registers for each block is reduced. It is necessary to choose and optimize appropriate number of threads for each SM (in a block).

The second storage type on GPU is shared memory. Shared memory is located in each stream processor and it is called on-chip or local memory. The latency of access to shared memory is longer than the latency of access to registers. Access data in shared memory takes about two clock cycles to perform one data fetch. Different from register, shared memory can be accessed by multiple threads in the same

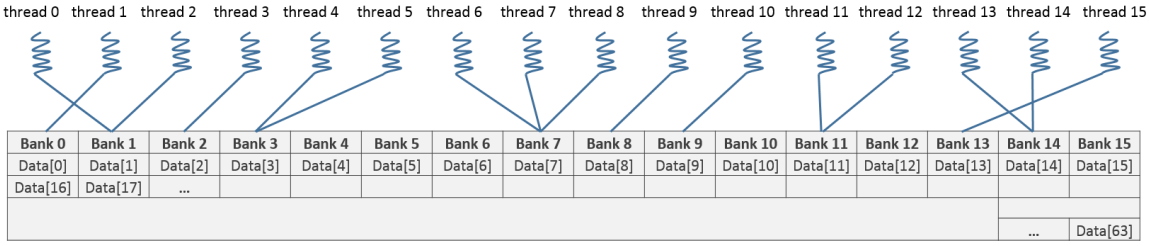
block. Therefore, shared memory plays significant roles in establishing communication among threads in a same block. If data needs to be read and write frequently, transfer data from global memory to shared memory is one solution to improve access latency.

While using shared memory in CUDA programming, bank conflict is a situation that increases the access time to local memory. In Nvidia GPUs, Shared memory is divided into memory banks. Thus, memory belongs to the same bank cannot be accessed at the same time. In Figure 4.4, each SM contains 16 load/store units, thus threads access memory as a group of 16, which means 16 threads access data in local memory at the same time. If threads access data that belongs to the same bank, bank conflict will occur. In Figure 4.5(b), there are multiple threads access bank1, bank3, bank7, and bank 14 at the same time. In this situation, bank conflicts occur, and theoretical access time will extend to 3 times more. Because thread 7, 8, 9 access data in the same bank, these accesses will be sequential. Since the memory that belongs to the same bank cannot be read or write at the same time, the access will be serialized instead of being executed in parallel. In order to maximum the access speed to shared memory, threads should access data that belongs to the different bank to avoid bank conflict. As shown in Figure 4.5(a), every thread in half warp access different back in a shared memory, and consequently no bank conflict will occur [58].

Global memory is the largest and most widely used memory in CUDA. It is further divided into global memory, texture memory, and constant memory. Global memory is located outside of SM (the DRAM in Figure 4.3 and the blue region in Figure 4.3), so it is also called off-chip memory. Access data in the global memory usually takes about 400-800 clock cycles. Therefore, Level-2 cache is used to reduce access time, because cores in a SM can access data in cache as fast as accessing data in register and shared memory. But if cache misses a hit, it still will take 400-800 clock cycles to fetch data in global memory.



(a) No Bank Conflict.



(b) Bank Conflict.

Fig. 4.5.: Bank Conflict.

4.4 CUDA Kernel Optimization

The algorithm of combinatorial map extraction is straightforward. Therefore, the thesis mainly focuses on optimize kernel on hardware. In order to optimize the combinatorial map extraction algorithm, the first improvement is to maximize the parallelization of the algorithm on the hardware. The extraction algorithm does not require much complex computation, it requires large amount of data modification like face removal, edge removal, and vertex removal. For this type of application, the bottlenecks of GPU program performance are memory bandwidth and access latency of the GPU global memory, which means the bottleneck is the time consumed by transferring the data from CPU memory to GPU memory (because during data transfer time, a CPU can finish very large percent of calculation) and time consumed by fetching data speed from GPU global memory into GPU kernel (access data in GPU global memory is slower than access data in CPU memory). To process 512×512×512 1-channel 8-bits images, it requires 128MB memory and on-chip memory does not have enough space to hold them. Therefore, data will be put in global memory and the

access to global memory will generate longer latency. To compensate latency and make CUDA program more efficient than CPU program, it is necessary to allocate enough amount of thread to process data at the same time.

maximizing the number of threads in a block is the option to maximum the number of threads running on the device. However, due to the limitation of hardware resource, it is necessary to estimate the number of registers used per threads, since using more registers than a SM have will reduce performance of kernel or even cause kernel launching failure. For GPU with computation capability 1.1 like GeForce GT 330M, each SM contains 8192 registers and the maximum simultaneous thread number per SM is 768. When 24 warps (threads number/ warp size = 768/32 = 24 warps) is activated on the SM, each thread can uses 10 registers during calculation. In this example, hardware occupancy reaches 100% leading to the best performance of GPU function. In another case, if one block contains 128 threads and each thread uses 15 registers. Then only 4 block can be activated on one SM (5 blocks use $5\times128\times15 = 9600$ registers that is larger than 8192, more than the hardware has). The hardware occupancy will be 66.67%. Therefore, to improve the performance of GPU, the threads number in a SM should be close to the number of maximum simultaneous threads and the number of registers per thread is smaller than the maximum register number of SM. Grid size can be calculated from block size [56]:

$$S_{block} = max(N_{reg}/N_{rpt}, N_{tmax}) \tag{4.2}$$

$$if(N_{data}\ mod\ S_{block} \neq 0)N_{block} = N_{data}/S_{block} + 1 \tag{4.3}$$

$$elseN_{block} = N_{data}/S_{block} \tag{4.4}$$

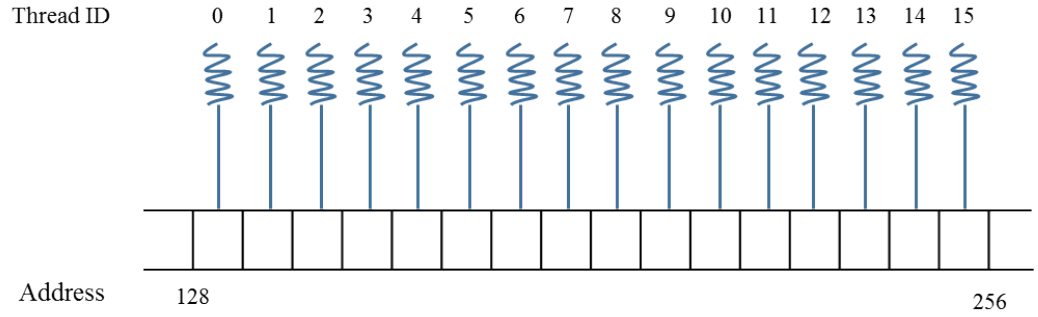
In the equation, S_{block}, N_{reg}, N_{rpt}, N_{tmax}, N_{data}, and N_{block} represents the number of threads per block, the number of register per block, the number of register per thread, the maximum simultaneous threads, the total number of data that needs to be processed, and number total block number correspondingly. In equation 4.2, the

the maximum number between N_{reg}/N_{rpt} and N_{tmax} is chosen to calculate S_{block} . Then S_{block} are substituted to equation 4.3 and 4.4 to calculate N_{block}. If N_{data} module S_{block} is equal to 0, then N_{block} is equal to the quotient of N_{data} divided by S_{block}. Otherwise, this number N_{block} is incremented by 1.

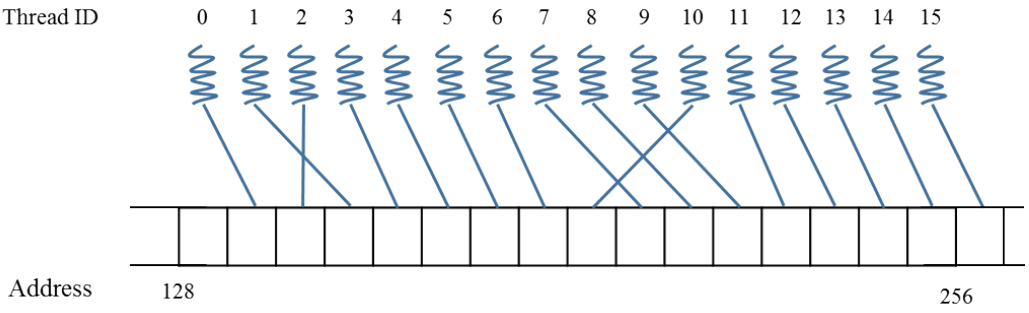
When analyzing GPU application performance, higher thread occupancy does not always indicates better performance. After threads occupancy exceeds 50%, the increase of thread occupancy cannot improve program performance significantly. This is because more threads in SM results in fewer resources per threads, and extra resources will be allocated on global memory, which consequently reduces program performance. Therefore, if thread occupancy reaches 50%, considering using shared memory and memory coalescing is an alternative method to improve the performance of CUDA program.

Another improvement is maximizing the data fetching speed in global memory by using access coalescing. Since combinatorial map data are stored in global memory, the speed of fetching input data to kernel and transferring output to memory is crucial for the performance of GPU function. GPU contains multiple memory controller units (MCU) that responsible for accessing GPU global memory, MCU occupancy is a critical factor of CUDA algorithm running time [53]. To ensure data can be fetched simultaneously, ensuring memory coalescing [53] is important for memory accessing performance. Memory that aligned with threads means multiple memory accesses are combined into one transition. The array that allocated in device is aligned to 256-byte memory segments by the CUDA driver. Coalescing access requires every thread in a warp reads or writes global memory is aligned to a memory address, which equals to multiplication of the number of elements in the array with the variable type of the array (for example, the Nth thread accesses integer array located at N $\times$ 4 byte) [59]. For any devices with computation capability (CC) 1.0 and 1.1, the Nth thread in a half-warp (16 threads) must access the Nth element in a memory segment. In our

application of aligned access, if an array of char is allocated at address 0X000000, then the address that accessed by a half-warp must be a multiple of 0X000010. In contrast, misaligned access will results in separate transaction and reduction of bandwidth (because 4 bytes are requested per transaction by load and store unit in SM, accessing array of 1-byte variable type results in the bandwidth is reduced by 4. For 4 byte variable type like array of integer, bandwidth reduced by 16. If data is not located in same memory segment, then it takes more separate transactions to load data in the different segments).



(a) aligned access.



(b) misaligned access.

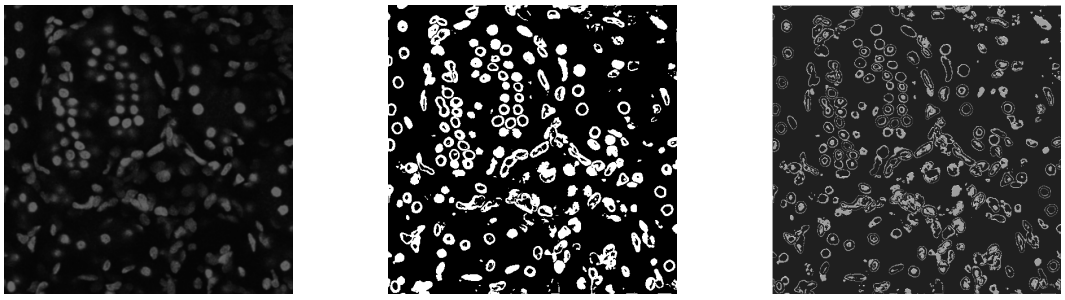
Fig. 4.6.: Thread Global Memory Access.

For CUDA device with computation capability of 1.2 and 1.3, CUDA introduces a new concept of aligned access. For any non-sequential data, CUDA driver will determine whether data are located in the same segment. If data belongs to the same segment, it only takes one access to load all non-sequential data as shown in Figure

4.6(a). For any consecutive data that located in two different segments, CUDA driver treats the data in the same segment as aligned data. It takes two accesses to load data from memory to SM. For any data that are not address aligned as shown in Figure 4.6(b), CUDA driver make 2 transactions to load all data (First transaction load data from 128 to 256 and second transaction load data from 256 to 384) and extra data will be discarded. For Nvidia GPU with computation capability higher than 2.0, the cost of unaligned access is negligible. Nvidia GPU with CC higher than 2.0 have L1 cache in each SM. The effect of unaligned memory access is negligible due to CDUA kernel combines memory accesses into cache line. Optimization technique of coalescing access is ignored on these type of GPUs.

4.5 Result

Figure 4.7(a) shows the a labeled cell image and Figure 4.7(c) is the corresponding combinatorial map using encoding method discussed in section 3.2.6 (using 7 bits to represents intervoxel elements). Image is pre-processed and pre-process result is shown in Figure 4.7(b). Image are filtered using Gaussian kernel with sigma = 2 initially and then filtered using sharpen kernel. If a threshold is applied to the image, a rough edge map is generated as shown in Figure 4.7(b). Edge map is used as merging criteria for the combinatorial map shown in Figure 4.7(c) .



(a) Grayscale Image. (b) Edge Map. (c) Combinatorial Map.

Fig. 4.7.: Combinatorial Map Encoding Image

Figure 4.8 gives a comparison between CPU program running time and CUDA program running time for same extraction algorithms. CUDA program is executed on a GeForce GTX 760 Ti with 1152 CUDA cores at 823 MHz. CPU program is executed on Intel(R) Core(i7) processor at frequency 2.93GHz. Using CPU program, face removal operation takes about 13.08s, edge removal operation takes 43.33s and vertex removal takes 18.86s. With parallel algorithm, running time of each operation reduced to 4.986s, 6.004s, 2.00s. CPU program takes 5 times more executing time to generate same combinatorial map compare to CUDA program.

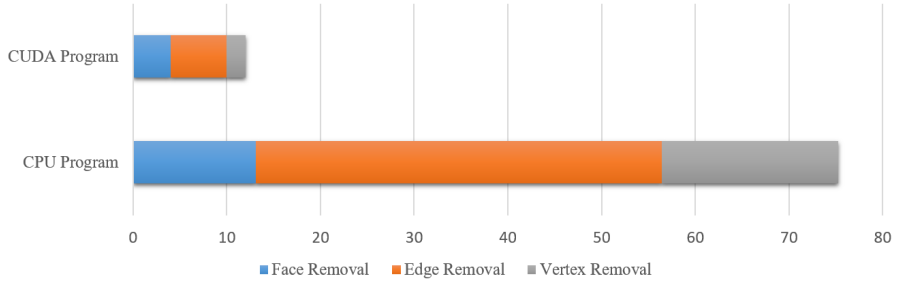


Fig. 4.8.: Running Time Comparison With CPU Algorithm and GPU Algorithm.

5. CONCLUSION AND FUTURE WORK

In second chapter of this thesis, we first talked about the background information about Voxx 3 and its software module composition. The detail of rendering module are discussed in detail. We introduced the procedure of a ray casting algorithm and explained how the stack images are stored in GPU texture memory. formula of accessing data inside GPU memory using normalized coordinate are deducted in this thesis. Then we discussed other modules in Voxx 3 including the parameter module and image processing module. In the last result section, representative rendering results with original voxel values as well as modified look-up value of ray casting algorithm are presented using Voxx 3.

In the third chapter, we first reviewed the history of graphs that widely used in segmentation algorithm and their benefits as well as drawbacks their application. The advantages of combinatorial map in GPU calculation is also explained. We further discussed the definition of combinatorial map and an algorithm to extracted combinatorial map and operation to remove faces, edges, and vertices. Then We provided its application of a 3D region-based segmentation using this map. Based on our application and limitation in running time, we proposed encoding methods of combinatorial map and its implementation in 3D region-based merging segmentation. In the result section, we analyzed time complicity of the 3D segmentation algorithm using the encoded combinatorial map proposed in previous sections. And provided some segmentation result and time cost using two different data sets.

In the implementation, we found that the running time of 3D segmentation increase quadratically as the data set increase. When data set is large, the algorithm is impractical is impractical for real-time applications. Therefore, in the forth chapter,

we discussed the NVidia Fermi GPU architecture and CUDA programming module. In order to improve CUDA kernel efficiency, several optimization strategies for a parallel algorithm of combinatorial map generation are proposed. we provided the combinatorial map operation in CUDA kernel and analyzed its time cost. At last, encoded map image and running time of extraction combinatorial map from a labeled stack image are provided in result section.

Now, we have seen that GPU process combinatorial map much faster than CPU. In future work, we want to modify the 3D segmentation algorithm in parallel to adapt combinatorial map extraction operation in order to improve running time cost. However, there are still some work that need to be considered in order to fully parallelize the 3D segmentation algorithm on GPU. The first consideration in the memory bandwidth that discussed in section 4.3. To completely utilize the computing power of GPU, we have to optimize the data inside GPU memory to increase the parallelization of cores accessing memory. We can achieve this by either avoid misaligned memory access or increase the probability of cache hit. In paper [60], authors proposed a z-order data set storage method to efficiently utilize memory access, increase cache hit rate, and coalesce memory access. The other technique to improve the algorithm efficiency is kernel occupancy. Maximize the kernel occupancy can keep the hardware work at its maximum potential. In section 4.4, we have discussed how to optimize thread number in a kernel. The next question is to modify and optimize the 3D segmentation algorithm on CUDA kernel, since the current algorithm is based on a greedy algorithm. The reason is that if is we still keep the idea of the greedy algorithm, the data parallel ratio decreased correspondingly, which means we did not fully utilize the hardware. Therefore, some modification has to be applied to the segmentation algorithm.

LIST OF REFERENCES

LIST OF REFERENCES

[1] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," *SIGGRAPH Comput. Graph.*, vol. 22, pp. 65–74, June 1988.

[2] S. D. Roth, "Ray casting for modeling solids," *Computer Graphics and Image Processing*, vol. 18, no. 2, pp. 109–144, 1982.

[3] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, "The volumepro real-time ray-casting system," 1999.

[4] P. S. Calhoun, B. S. Kuszyk, D. G. Heath, J. C. Carley, and E. K. Fishman, "Three-dimensional volume rendering of spiral ct data: Theory and method," *RadioGraphics*, vol. 19, no. 3, pp. 745–764, 1999.

[5] W. Bang, J. Li, B. Yan, C. Tong, and J. Chen, "Ray casting algorithm based on improved empty voxel skipping method," *Computer Engineering*, vol. 38, no. 2, pp. 264–266, 2012.

[6] E. Gobbetti, F. Marton, and J. A. Iglesias Guitin, "A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets," *The Visual Computer*, vol. 24, no. 7-9, pp. 797–806, 2008.

[7] W. F. Bronsvoort, J. J. van Wijk, and F. W. Jansen, "Two methods for improving the efficiency of ray casting in solid modelling," *Computer-Aided Design*, vol. 16, no. 1, pp. 51–55, 1984.

[8] W. M. Hsu, "Segmented ray casting for data parallel volume rendering," in *Proceedings of the 1993 Symposium on Parallel Rendering*, PRS '93, (New York, NY, USA), pp. 7–14, ACM, 1993.

[9] M. Hadwiger, C. Sigg, H. Scharsach, K. Bhler, and M. Gross, "Real-time ray-casting and advanced shading of discrete isosurfaces," *Computer Graphics Forum*, vol. 24, no. 3, pp. 303–312, 2005.

[10] B. Mora, J. P. Jessel, and R. Caubet, "A new object-order ray-casting algorithm," in *Visualization, 2002. VIS 2002. IEEE*, pp. 203–210, Nov 2002.

[11] H. Ray, H. Pfister, D. Silver, and T. A. Cook, "Ray casting architectures for volume visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, pp. 210–223, July 1999.

[12] R. Osborne, H. Pfister, H. Lauer, N. McKenzie, S. Gibson, W. Hiatt, and T. Ohkami, "Em-cube: An architecture for low-cost real-time volume rendering," pp. 131–138, 08/1997 1997.

[13] H. Pfister and A. Kaufman, "Cube-4—a scalable architecture for real-time volume rendering," in *Proceedings of the 1996 Symposium on Volume Visualization*, VVS '96, (Piscataway, NJ, USA), pp. 47–ff., IEEE Press, 1996.

[14] T. J. Cullip and U. Neumann, "Accelerating volume reconstruction with 3d texture hardware," tech. rep., Chapel Hill, NC, USA, 1994.

[15] J. Groff, "An intro to modern opengl. chapter 1: The graphics pipeline." http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html, 2016. [Online; Last Date Accessed: 25-March-2016].

[16] OpenGL, "Tessellation." https://www.opengl.org/wiki/Tessellation, 2016. [Online; Last Date Accessed: 25-March-2016].

[17] A. Williams, S. Barrus, R. K. Morley, and P. Shirley, "An efficient and robust ray-box intersection algorithm," in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), ACM, 2005.

[18] A. Williams, S. Barrus, R. K. Morley, and P. Shirley, "An efficient and robust ray-box intersection algorithm," in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), ACM, 2005.

[19] F. Dunn and I. Parberry, *3D Math Primer for Graphics and Game Development*. Wordware game math library, Wordware Pub., 2002.

[20] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, pp. 679–698, Nov 1986.

[21] J.-P. Braquelaire and L. Brun, "Image segmentation with topological maps and inter-pixel representation," *Journal of Visual Communication and Image Representation*, vol. 9, no. 1, pp. 62–79, 1998.

[22] A. Dupas and G. Damiand, *First Results for 3D Image Segmentation with Topological Map*, vol. 4992 of *Lecture Notes in Computer Science*, book section 45, pp. 507–518. Springer Berlin Heidelberg, 2008.

[23] P. F. Felzenszwalb and D. P. Huttenlocher, "Image segmentation using local variation," in *Computer Vision and Pattern Recognition, 1998, IEEE Computer Society Conference, 1998*, pp. 98–104.

[24] G. Damiand, "Topological model for 3d image representation: Definition and incremental extraction algorithm," *Computer Vision and Image Understanding*, vol. 109, no. 3, pp. 260–289, 2008.

[25] P. Lienhardt, "Topological models for boundary representation: a comparison with n-dimensional generalized maps," *Comput. Aided Des.*, vol. 23, no. 1, pp. 59–82, 1991.

[26] A. Rosenfeld, "Adjacency in digital pictures," *Information and Control*, vol. 26, no. 1, pp. 24–33, 1974.

[27] D. Willersinn and W. G. Kropatsch, "Dual graph contraction for irregular pyramids," in *Pattern Recognition, 1994. Vol. 3 - Conference C: Signal Processing of the 12th IAPR International Conference*, pp. 251–256 vol.3.

[28] W. G. Kropatsch, "Building irregular pyramids by dual-graph contraction," *Vision, Image and Signal Processing, IEEE Proceedings* -, vol. 142, no. 6, pp. 366–374, 1995.

[29] S. Biasotti, D. Giorgi, M. Spagnuolo, and B. Falcidieno, "Reeb graphs for shape analysis and applications," *Theoretical Computer Science*, vol. 392, no. 13, 2008. Computational Algebraic Geometry and Applications.

[30] J. G. Pailloncy and J. M. Jolion, *The Frontier-Region Graph*, vol. 12 of *Computing Supplement*, book section 13, pp. 123–134. Springer Vienna, 1998.

[31] Y. Bertrand, C. Fiorio, and Y. Pennaneach, *Border Map: A Topological Representation for nD Image Analysis*, vol. 1568 of *Lecture Notes in Computer Science*, book section 19, pp. 242–257. Springer Berlin Heidelberg, 1999.

[32] S. Suzuki and K. be, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32–46, 1985.

[33] Y. Bertrand, G. Damiand, and C. Fiorio, "Topological map: Minimal encoding of 3d segmented images," *Proc. of 3rd Workshop on Graph-Based Representation in Pattern Recognition (GBR)*, no. 2001, pp. 64–73.

[34] A. Rosenfeld, "Three-dimensional digital topology," *Information and Control*, vol. 50, no. 2, pp. 119–127, 1981.

[35] E. Antnez, R. Marfil, and A. Bandera, "Combining boundary and region features inside the combinatorial pyramid for topology-preserving perceptual image segmentation," *Pattern Recognition Letters*, vol. 33, no. 16, pp. 2245–2253, 2012.

[36] F. Baldacci, A. Braquelaire, P. Desbarats, and J.-P. Domenger, *3D Image Topological Structuring with an Oriented Boundary Graph for Split and Merge Segmentation*, vol. 4992 of *Lecture Notes in Computer Science*, book section 48, pp. 541–552. Springer Berlin Heidelberg, 2008.

[37] F. Baldacci, A. Braquelaire, and G. Damiand, *3D Topological Map Extraction from Oriented Boundary Graph*, vol. 5534 of *Lecture Notes in Computer Science*, book section 29, pp. 283–292. Springer Berlin Heidelberg, 2009.

[38] S. Chmutov, "Generalized duality for graphs on surfaces and the signed bollobsriordan polynomial," *Journal of Combinatorial Theory, Series B*, vol. 99, no. 3, pp. 617 – 638, 2009.

[39] G. Damiand and P. Resch, "Topological map based algorithms for 3d image segmentation," in *Proceedings of the 10th International Conference on Discrete Geometry for Computer Imagery*, DGCI '02, (London, UK, UK), pp. 220–231, Springer-Verlag, 2002.

[40] J. C. Tilton, "Image segmentation by iterative parallel region growing and splitting," in *Geoscience and Remote Sensing Symposium, 1989. IGARSS'89. 12th Canadian Symposium on Remote Sensing., 1989 International*, vol. 4, pp. 2420–2423.

[41] J. C. Tilton, "Image segmentation by iterative parallel region growing with applications to data compression and image analysis," in *2nd Symposium on the Frontiers of Massively Parallel Computation, 1988.*, pp. 357–360.

[42] L. Brun, J. P. Domenger, and J. P. Braquelaire, *Discrete Maps: a Framework for Region Segmentation Algorithms*, vol. 12 of *Computing Supplement*, book section 9, pp. 83–92. Springer Vienna, 1998.

[43] G. Damiand and P. Resch, "Split-and-merge algorithms defined on topological maps for 3d image segmentation," *Graph. Models*, vol. 65, no. 1-3, pp. 149–167, 2003.

[44] M. Unger, T. Pock, and H. Bischof, "Continuous globally optimal image segmentation with local constraints," *Computer Vision Winter Workshop*, vol. 2008, 2008.

[45] Y. Bertrand, G. Damiand, and C. Fiorio, "Topological encoding of 3d segmented images," in *Discrete Geometry for Computer Imagery* (G. Borgefors, I. Nystrm, and G. di Baja, eds.), vol. 1953 of *Lecture Notes in Computer Science*, pp. 311–324, Springer Berlin Heidelberg, 2000.

[46] M. Hopf and T. Ertl, "Accelerating 3d convolution using graphics hardware," in *Visualization '99. Proceedings*, pp. 471–564.

[47] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical image processing on the gpu past, present and future," *Medical Image Analysis*, vol. 17, no. 8, pp. 1073–1094, 2013.

[48] R. Shams, P. Sadeghi, R. A. Kennedy, and R. I. Hartley, "A survey of medical image registration on multicore and the gpu," *Signal Processing Magazine, IEEE*, vol. 27, no. 2, pp. 50–60, 2010.

[49] P. Lei, G. Lixu, and X. Jianrong, "Implementation of medical image segmentation in cuda," in *Information Technology and Applications in Biomedicine, 2008. ITAB 2008. International Conference on*, pp. 82–85.

[50] S. A. Mahmoudi, F. Lecron, P. Manneback, M. Benjelloun, and S. Mahmoudi, "Gpu-based segmentation of cervical vertebra in x-ray images," in *2010 IEEE International Conference on Cluster Computing Workshops and Posters (Cluster Workshops)*, pp. 1–8.

[51] J. Wassenberg, W. Middelmann, and P. Sanders, *An Efficient Parallel Algorithm for Graph-Based Image Segmentation*, vol. 5702 of *Lecture Notes in Computer Science*, book section 122, pp. 1003–1010. Springer Berlin Heidelberg, 2009.

[52] A. Khotanzad and A. Bouarfa, "Image segmentation by a parallel, non-parametric histogram based clustering algorithm," *Pattern Recognition*, vol. 23, no. 9, pp. 961–973, 1990.

[53] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008.*, pp. 836–838.

[54] D. Lee, I. Dinov, B. Dong, B. Gutman, I. Yanovsky, and A. W. Toga, "Cuda optimization strategies for compute- and memory-bound neuroimaging algorithms," *Computer Methods and Programs in Biomedicine*, vol. 106, no. 3, pp. 175–187, 2012.

[55] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," 2008.

[56] D. Chou, "Cuda kernel optimization technique," in *GPGPU programming technology - from GLSL, CUDA, to OpenCL*, ch. 10, pp. 266–290, China Machine Press, 2011.

[57] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[58] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar, "GPGPU processing in CUDA architecture," *CoRR*, vol. abs/1202.4347, 2012.

[59] M. Harris, "How to access global memory efficiently in cuda c/c++ kernels." http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels, 2010. [Online; Last Date Accessed: 27-January-2016].

[60] H. Cho and Y. H. Kim, "Ray-casting algorithm and its considerations for parallel processing optimization techniques for parallel ray-casting algorithm," in *Integrated Circuits (ISIC), 2014 14th International Symposium on*, pp. 107–110, December 2014.

APPENDIX

A. APPENDIX

Table A.1: 3-Sewn Operation Look-up Table

Front	$(i, j, k-1)$	d_0 - d_8	d_1 - d_{11}	d_2 - d_{10}	d_3 - d_9
Right	$(i+1, j, k)$	d_{12} - d_4	d_{13} - d_7	d_{14} - d_6	d_{15} - d_5
Back	$(i, j, k+1)$	d_8 - d_0	d_{11} - d_1	d_{10} - d_2	d_9 - d_3
Left	$(i-1, j, k)$	d_4 - d_{12}	d_5 - d_{15}	d_6 - d_{14}	d_7 - d_{13}
Up	$(i, j-1, k)$	d_{16} - d_{22}	d_{17} - d_{21}	d_{18} - d_{20}	d_{19} - d_{23}
Bottom	$(i, j+1, k)$	d_{20} - d_{19}	d_{21} - d_{17}	d_{22} - d_{16}	d_{23} - d_{19}

Table A.2: 2-Sewn Operation Look-up Table

Front	$(i,j,k) == (i,j,k-1)$	d_5 - d_7	d_{15} - d_{13}	d_{18} - d_{16}	d_{20} - d_{22}
	$(i,j,k)! = (i,j,k-1)$	d_5 - d_3	d_{15} - d_1	d_{18} - d_0	d_{20} - d_2
Right	$(i,j,k) == (i+1,j,k)$	d_3 - d_1	d_9 - d_{11}	d_{19} - d_{17}	d_{23} - d_{21}
	$(i,j,k)! = (i+1,j,k)$	d_3 - d_5	d_9 - d_{17}	d_{19} - d_4	d_{23} - d_6
Back	$(i,j,k) == (i,j,k+1)$	d_7 - d_5	d_{13} - d_{15}	d_{16} - d_{18}	d_{22} - d_{20}
	$(i,j,k)! = (i,j,k+1)$	d_7 - d_9	d_{13} - d_{11}	d_{16} - d_8	d_{22} - d_{10}
Left	$(i,j,k) == (i-1,j,k)$	d_1 - d_3	d_{11} - d_9	d_{17} - d_{19}	d_{21} - d_{23}
	$(i,j,k)! = (i-1,j,k)$	d_1 - d_{15}	d_{11} - d_{13}	d_{17} - d_{12}	d_{21} - d_{14}
Up	$(i,j,k) == (i,j-1,k)$	d_0 - d_2	d_4 - d_6	d_8 - d_{10}	d_{12} - d_{14}
	$(i,j,k)! = (i,j-1,k)$	d_0 - d_{18}	d_4 - d_{19}	d_8 - d_{16}	d_{12} - d_{17}
Bottom	$(i,j,k) == (i,j+1,k)$	d_2 - d_0	d_6 - d_4	d_{10} - d_9	d_{14} - d_{12}
	$(i,j,k)! = (i,j+1,k)$	d_2 - d_{20}	d_6 - d_{23}	d_{10} - d_{22}	d_{14} - d_{21}

Table A.3: 1-Sewn Operation Look-up Table

d_0	$(i,j,k)d_3$	$(i+1,j,k)d_0$	$(i+1,j-1,k)d_1$	$(i,j-1,k)d_2$
d_1	$(i,j,k)d_0$	$(i,j-1,k)d_1$	$(i-1,j-1,k)d_2$	$(i-1,j,k)d_3$
d_2	$(i,j,k)d_1$	$(i,j-1,k)d_2$	$(i-1,j+1,k)d_3$	$(i,j+1,k)d_0$
d_3	$(i,j,k)d_2$	$(i,j+1,k)d_3$	$(i,j+1,k+1)d_0$	$(i+1,j,k)d_1$
d_4	$(i,j,k)d_7$	$(i,j,k+1)d_4$	$(i,j-1,k-1)d_5$	$(i,j-1,k)d_6$
d_5	$(i,j,k)d_4$	$(i,j-1,k)d_5$	$(i,j-1,k-1)d_6$	$(i,j,k-1)d_7$
d_6	$(i,j,k)d_5$	$(i,j,k-1)d_6$	$(i,j-1,k-1)d_7$	$(i,j-1,k)d_4$
d_7	$(i,j,k)d_6$	$(i,j+1,k)d_7$	$(i,j-1,k+1)d_4$	$(i,j,k+1)d_5$
d_8	$(i,j,k)d_{11}$	$(i-1,j,k)d_8$	$(i-1,j-1,k)d_9$	$(i,j-1,k)d_{10}$
d_9	$(i,j,k)d_{10}$	$(i,j-1,k)d_{11}$	$(i-1,j+1,k)d_8$	$(i-1,j,k)d_9$
d_{10}	$(i,j,k)d_9$	$(i+1,j,k)d_{10}$	$(i+1,j+1,k)d_{11}$	$(i,j+1,k)d_8$
d_{11}	$(i,j,k)d_8$	$(i,j-1,k)d_9$	$(i+1,j-1,k)d_{10}$	$(i+1,j,k)d_{11}$
d_{12}	$(i,j,k)d_{15}$	$(i,j,k-1)d_{12}$	$(i,j-1,k-1)d_{13}$	$(i,j-1,k)d_{14}$
d_{13}	$(i,j,k)d_{12}$	$(i,j-1,k)d_{13}$	$(i,j-1,k+1)d_{14}$	$(i,j,k+1)d_{15}$
d_{14}	$(i,j,k)d_{13}$	$(i,j,k+1)d_{14}$	$(i,j+1,k+1)d_{15}$	$(i,j+1,k)d_{12}$
d_{15}	$(i,j,k)d_{14}$	$(i,j+1,k)d_{15}$	$(i,j+1,k-1)d_{12}$	$(i,j,k-1)d_{13}$
d_{16}	$(i,j,k)d_{19}$	$(i+1,j,k)d_{16}$	$(i+1,j,k+1)d_{17}$	$(i,j,k+1)d_{18}$
d_{17}	$(i,j,k)d_{16}$	$(i,j,k+1)d_{17}$	$(i-1,j,k+1)d_{18}$	$(i-1,j,k)d_{19}$
d_{18}	$(i,j,k)d_{17}$	$(i-1,j,k)d_{18}$	$(i-1,j,k-1)d_{19}$	$(i,j,k-1)d_{16}$
d_{19}	$(i,j,k)d_{18}$	$(i,j,k-1)d_{19}$	$(i+1,j,k-1)d_{16}$	$(i+1,j,k)d_{17}$
d_{20}	$(i,j,k)d_{23}$	$(i+1,j,k)d_{20}$	$(i+1,j,k-1)d_{21}$	$(i,j,k-1)d_{22}$
d_{21}	$(i,j,k)d_{20}$	$(i,j,k-1)d_{21}$	$(i-1,j,k-1)d_{22}$	$(i-1,j,k)d_{23}$
d_{22}	$(i,j,k)d_{21}$	$(i-1,j,k)d_{22}$	$(i-1,j,k+1)d_{23}$	$(i,j,k+1)d_{20}$
d_{23}	$(i,j,k)d_{22}$	$(i,j,k+1)d_{23}$	$(i+1,j,k+1)d_{20}$	$(i+1,j,k)d_{21}$

Table A.4: 0-Sewn Operation Look-up Table

d_0	$(i,j,k)d_1$	$(i-1,j,k)d_0$	$(i-1,j-1,k)d_3$
d_1	$(i,j,k)d_2$	$(i,j+1,k)d_1$	$(i-1,j+1,k)d_0$
d_2	$(i,j,k)d_3$	$(i+1,j,k)d_2$	$(i+1,j+1,k)d_1$
d_3	$(i,j,k)d_0$	$(i,j-1,k)d_3$	$(i+1,j-1,k)d_2$
d_4	$(i,j,k)d_5$	$(i,j,k-1)d_4$	$(i,j-1,k+1)d_7$
d_5	$(i,j,k)d_6$	$(i,j+1,k)d_5$	$(i,j+1,k-1)d_4$
d_6	$(i,j,k)d_7$	$(i,j,k+1)d_6$	$(i,j-1,k+1)d_5$
d_7	$(i,j,k)d_4$	$(i,j-1,k)d_7$	$(i,j+1,k+1)d_6$
d_8	$(i,j,k)d_9$	$(i+1,j,k)d_8$	$(i+1,j-1,k)d_{11}$
d_9	$(i,j,k)d_8$	$(i-1,j+1,k)d_{11}$	$(i-1,j-1,k)d_{10}$
d_{10}	$(i,j,k)d_{11}$	$(i-1,j,k)d_{10}$	$(i-1,j+1,k)d_9$
d_{11}	$(i,j,k)d_{10}$	$(i,j+1,k)d_9$	$(i+1,j+1,k)d_8$
d_{12}	$(i,j,k)d_{13}$	$(i,j,k+1)d_{12}$	$(i,j-1,k+1)d_{15}$
d_{13}	$(i,j,k)d_{14}$	$(i,j+1,k)d_{13}$	$(i,j+1,k+1)d_{12}$
d_{14}	$(i,j,k)d_{15}$	$(i,j,k-1)d_{14}$	$(i,j+1,k-1)d_{13}$
d_{15}	$(i,j,k)d_{12}$	$(i,j-1,k)d_{15}$	$(i,j-1,k-1)d_{14}$
d_{16}	$(i,j,k)d_{17}$	$(i-1,j,k)d_{16}$	$(i-1,j,k+1)d_{19}$
d_{17}	$(i,j,k)d_{18}$	$(i,j,k-1)d_{17}$	$(i+1,j,k-1)d_{16}$
d_{18}	$(i,j,k)d_{19}$	$(i+1,j,k)d_{18}$	$(i+1,j,k-1)d_{17}$
d_{19}	$(i,j,k)d_{16}$	$(i,j,k+1)d_{19}$	$(i+1,j,k+1)d_{18}$
d_{20}	$(i,j,k)d_{21}$	$(i-1,j,k)d_{20}$	$(i-1,j,k-1)d_{23}$
d_{21}	$(i,j,k)d_{22}$	$(i,j,k+1)d_{21}$	$(i-1,j,k+1)d_{20}$
d_{22}	$(i,j,k)d_{23}$	$(i+1,j,k)d_{22}$	$(i+1,j,k+1)d_{21}$
d_{23}	$(i,j,k)d_{20}$	$(i,j,k-1)d_{23}$	$(i+1,j,k-1)d_{22}$